1. **Node.js**: Node.js is an open-source, cross-platform JavaScript runtime environment that allows you to run JavaScript code outside of a web browser. It uses the V8 JavaScript engine from Google Chrome and provides various built-in modules for interacting with the file system, networking, and more. Node.js enables developers to build server-side applications, command-line tools, and even desktop applications using JavaScript.
2. **npm (Node Package Manager)**: npm is the default package manager for Node.js. It's a command-line tool that allows developers to discover, install, and manage dependencies (external libraries or modules) for their Node.js projects. With npm, developers can easily install packages from the npm registry (a public repository of JavaScript packages), publish their own packages, and manage project dependencies through a **package.json** file.
3. **npx**: npx is a command-line tool that comes bundled with npm starting from version 5.2.0. It stands for "Node Package Runner" and is used to execute Node.js packages directly without needing to install them globally or locally. npx allows you to run command-line tools or scripts from npm packages, either from the npm registry or from your local **node\_modules** folder, in a hassle-free way. It helps to avoid polluting your global or local package installation with unnecessary packages and ensures that you're running the latest version of a package, even if it's not installed on your system.

In simple terms:

* **Node.js**: Allows you to run JavaScript code outside of a web browser, enabling server-side programming and other types of applications using JavaScript.
* **npm**: Helps you manage dependencies and packages for your Node.js projects, making it easy to install, publish, and share code with other developers.
* **npx**: Lets you run command-line tools or scripts from npm packages without installing them globally or locally, providing a convenient way to execute Node.js packages without worrying about installation or version conflicts.

Top of Form

PROCESS:

Mkdir server

Npm I -g nodemon

(g means global installation, nodemon automatically restarts server when changes detected. If we didn’t include g it would only install it within our current directory)

Packages installed using npm:

Body-parser: parses through request body

Bcrypt: password encryption

Cors: cross origin requests

Dotenv: environment variables

Gridfs-stream: file upload

Multer, multer-gridfs-storage: for uploading files locally

Helmet: request safety, morgan: login, jsonwebtoken: authentication

Mongoose: mongodb access

The **npm init -y** command is a shortcut for initializing a new npm project with default settings without prompting for input. Here's what each part of the command means:

* **npm**: This is the command-line tool used for managing Node.js packages.
* **init**: This is a subcommand of npm used to initialize a new npm project. It prompts you to enter information about your project, such as its name, version, description, entry point, author, license, etc.
* **-y**: This is a shortcut for the **--yes** option. When you provide **-y** or **--yes** with the **npm init** command, it automatically accepts the default values for all prompts, without asking for user input. It's useful for quickly initializing a new project with default settings, especially in automated or scripted environments.

So, when you run **npm init -y**, npm creates a new **package.json** file in the current directory with default settings without prompting you for any input. It's a convenient way to quickly create a new npm project when you're in a hurry or don't need to customize the project settings.

**npm init -y** is a command used to quickly initialize a new Node.js project with default values for the **package.json** file without requiring user input. The **-y** flag stands for "yes," indicating that you want to accept all default values provided by npm without prompting for confirmation.

The **package.json** file is a crucial part of any Node.js project. It serves several purposes:

1. **Project Metadata**: **package.json** contains metadata about the project, such as the project name, version, description, author, and license information. This metadata helps developers and tools understand the project's purpose and how it's structured.
2. **Dependencies Management**: One of the most important roles of **package.json** is to manage project dependencies. It lists all the dependencies required by the project, including their names and versions. When someone else clones or downloads your project, they can run **npm install**, and npm will install all the dependencies listed in **package.json**.
3. **Scripts**: **package.json** allows you to define scripts that can be executed using **npm run <script-name>**. These scripts can automate common tasks like building the project, running tests, or deploying the application.
4. **Configuration**: You can use **package.json** to store configuration settings for your project or tools. For example, you can define ESLint rules, Babel presets, or TypeScript configurations within **package.json**

Dependencies-> external packages

In package.json, “main”:”index.js” means that the entry point of the project is index.js. So, when someone installs your package as a dependency in their project using npm (e.g., **npm install your-package**), npm looks for the file specified in the **"main"** field and loads it. In this case, it will look for a file named **index.js** within the root directory of your package.

Do this in the server directory, package json is created after this command

"type":"module", is added in package.json file so that we can use import

MIDDLEWARE

Middleware configurations, particularly in the context of web development, refer to settings or configurations applied to middleware components within a web framework. Middleware acts as a bridge between incoming HTTP requests and outgoing HTTP responses in a web application. It allows developers to perform additional processing, such as authentication, logging, error handling, and more, before or after handling the request.

Middleware configurations typically involve setting up and configuring middleware components according to the specific requirements of the application. These configurations may include:

1. **Middleware Registration**: Registering middleware components with the web framework to ensure they are invoked during request processing.
2. **Ordering**: Specifying the order in which middleware components should be executed. Middleware execution order can affect the behavior of the application, especially when one middleware component depends on the output of another.
3. **Middleware Parameters**: Configuring middleware components with parameters or options to customize their behavior. For example, specifying authentication credentials for an authentication middleware.
4. **Error Handling**: Configuring error-handling middleware to handle errors or exceptions that occur during request processing.
5. **Route-specific Middleware**: Configuring middleware to apply only to specific routes or groups of routes within the application.
6. **Conditional Middleware**: Setting up middleware to execute conditionally based on certain criteria, such as request headers, query parameters, or authentication status.
7. **Middleware Stacks**: Organizing middleware components into stacks or pipelines to streamline request processing and improve code maintainability.
8. **Third-party Middleware**: Integrating and configuring third-party middleware components provided by libraries or packages to extend the functionality of the application.

Overall, middleware configurations play a crucial role in defining the behavior and functionality of a web application. By carefully configuring middleware components, developers can implement various cross-cutting concerns, enforce application-wide policies, and enhance the security, performance, and reliability of their applications.

NOTE:  
import bodyParser from “body-parser”; here bodyParser is not a class but it is a MIDDLEWARE FUNCTION. Middleware functions in Express.js are functions that have access to the request object (**req**), the response object (**res**), and the next middleware function in the application's request-response cycle. Middleware functions can perform tasks such as parsing request bodies, logging requests, authenticating users, etc.

EXPRESS JS

Express.js is a minimal and flexible web application framework for Node.js, designed for building web applications and APIs. It provides a robust set of features for web and mobile applications, including routing, middleware support, template engines, static file serving, and more. Express.js simplifies the process of building web servers and handling HTTP requests and responses in Node.js by providing a high-level abstraction layer over the built-in Node.js HTTP module.

Key features of Express.js include:

1. **Routing**: Express.js provides a simple and intuitive API for defining routes based on HTTP methods (GET, POST, PUT, DELETE, etc.) and URL patterns. Routes are used to handle incoming HTTP requests and define the application's behavior based on the requested URL and method.
2. **Middleware**:
3. Middleware functions in Express.js are typically defined with three parameters: **req**, **res**, and **next**. These parameters represent the request object (**req**), the response object (**res**), and a function (**next**) that passes control to the next middleware function in the application's request-response cycle.
4. When Express.js processes an incoming HTTP request, it sequentially invokes each middleware function registered with the application. Each middleware function is called with the **req** and **res** objects representing the current request and response, respectively. This allows middleware functions to inspect and modify the request and response objects as needed during request processing.
5. For example, a middleware function might modify the response object to set HTTP headers, send a response to the client, or pass control to the next middleware function by calling the **next** function. By having access to the **res** object, middleware functions can customize the behavior of the application and implement various functionalities, such as logging, authentication, response compression, and more, in a modular and reusable way.

ROUTE HANDLER

Sure! In Express.js, a route handler is a callback function that gets executed when a specific route is matched during an incoming HTTP request. It's essentially the code that runs when a request matches a particular URL and HTTP method combination.

Here's a breakdown of the concept:

1. **Defining Routes**: In Express.js, you define routes using methods like **app.get()**, **app.post()**, **app.put()**, **app.delete()**, etc. These methods take two arguments: the route path (a string representing the URL pattern) and the route handler (a callback function).
2. **Route Handler Function**: The route handler function is the second argument passed to the route methods. It's a function that takes two arguments: **req** (the request object) and **res** (the response object). The route handler function is responsible for handling the incoming request and generating an appropriate response.
3. **Execution**: When an incoming request is received by the Express application, the Express router goes through the defined routes and matches the request's method and URL pattern to the appropriate route handler. If a match is found, the corresponding route handler function is executed.
4. **Request and Response Objects**: Inside the route handler function, you have access to the **req** and **res** objects, which represent the current request and response, respectively. You can use these objects to access information about the incoming request (such as parameters, query strings, headers, etc.) and to send back a response to the client.

const express = require('express');

const app = express();

// Define a route handler for GET requests to the root URL

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

res.send('Hello, World!');

});

// Define a route handler for POST requests to the '/submit' URL

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

res.send('Form submitted successfully!');

});

// Define a route handler for PUT requests to the '/update/:id' URL

app.put('/update/:id', (req, res) => {

const id = req.params.id;

res.send(`Updated resource with ID ${id}`);

});

// Define a route handler for DELETE requests to the '/delete/:id' URL

app.delete('/delete/:id', (req, res) => {

const id = req.params.id;

res.send(`Deleted resource with ID ${id}`);

});

// Start the Express server

app.listen(3000, () => {

console.log('Server is running on port 3000');

});

In the above code:

* **'/products'** and **'/login'** are the specified URLs.
* When a client makes a GET request to **/products** or a POST request to **/login**, the corresponding route handlers will be executed.
* These URLs are not like hyperlinks you click on in a web browser. They represent the paths that clients can use to access specific resources or perform specific actions on your server.

So, when I mention "specified URLs", it's in the context of these route paths that you define in your Express application to handle incoming HTTP requests.

**app.get()** is used to define a route handler for HTTP GET requests.

The line **const app = express();** in Node.js creates an instance of the Express.js framework and assigns it to the variable **app**.

Here's a breakdown:

* **express()** is a function provided by the Express.js framework. When called, it returns a new Express application object.
* **const app** declares a constant variable named **app** to store the Express application object. Using **const** ensures that the variable **app** cannot be reassigned to a different value elsewhere in the code. This is a common practice in JavaScript to prevent accidental reassignment of important variables.

DOTENV

**.env** files are simple text files commonly used to store configuration variables for your application. They are typically used in Node.js and other server-side environments to keep sensitive or environment-specific configuration separate from your codebase.

Here's how **.env** files work:

1. **Key-Value Pairs**: **.env** files consist of key-value pairs, where each line represents one variable assignment. For example:

**.env** files are particularly useful for storing sensitive information like API keys, database credentials, or other configuration variables that may vary between different environments (such as development, staging, and production). Using **.env** files helps keep this sensitive information out of your codebase and provides an easy way to manage configuration across different environments.

dotenv.config()

1. **Parsing the File**: If the **.env** file exists, **dotenv.config()** parses the file and loads the key-value pairs as environment variables into the Node.js **process.env** object.
2. **Accessing Environment Variables**: After calling **dotenv.config()**, you can access the environment variables defined in the **.env** file using **process.env.VARIABLE\_NAME**, where **VARIABLE\_NAME** is the name of the variable defined in the **.env** file.

const app=express(); app.use(express.json());

1. **Middleware Function**: **express.json()** is a middleware function provided by Express.js.
2. **Parsing JSON**: When you use **app.use(express.json())**, you're telling your Express application to use this middleware for every incoming request.
3. **JSON Parsing**: When an incoming request with a JSON payload (such as a POST request with JSON data in the body) is received, **express.json()** parses the JSON data and exposes it on **req.body**.

INDEX.JS

In web development projects, the **index.js** file is often used as the entry point or main file for a Node.js application or a particular module within a larger project. Its specific use can vary depending on the project's structure and the developer's preferences, but here are some common purposes:

1. **Server Initialization**: In a Node.js application, **index.js** may be used to initialize and start the server. It's common to set up routing, middleware, and other server configurations in this file.
2. **Module Entry Point**: In modular JavaScript projects, **index.js** can serve as the entry point for a particular module or component. Other files within the module may import or require the module's functionality from **index.js**.
3. **Exporting Module Functionality**: **index.js** can serve as a central location for exporting functions, classes, or other modules from within a directory or package. This can make it easier for other parts of the project to import and use the exported functionality.
4. **Default File Name**: Some tools and frameworks, such as Create React App for React.js projects, automatically generate an **index.js** file as the default entry point for the application. In this case, it's used to render the root component of the application.
5. **Folder Structure Convention**: In many projects, using **index.js** as the main file within directories is a common convention for maintaining a clear and consistent folder structure. It serves as a recognizable starting point when navigating through the project's codebase.

Overall, **index.js** serves various purposes in web development projects, but its primary role is often as an entry point for starting an application, initializing modules, exporting functionality, or following folder structure conventions.

App.use()

Yes, you're correct! In Express.js, **app.use()** is a method used to mount middleware functions in the application's middleware stack. Middleware functions are functions that have access to the request object (**req**), the response object (**res**), and the next middleware function in the application’s request-response cycle.

Here's what **app.use()** does:

1. **Mounting Middleware**: **app.use()** is used to mount middleware functions in the Express application. Middleware functions can be used to perform various tasks such as parsing request bodies, handling authentication, logging requests, setting headers, and more.
2. **Order of Execution**: Middleware functions are executed sequentially in the order they are defined. When a request is received, Express executes each middleware function in the middleware stack, one after the other, until one of the middleware functions sends a response or calls the **next()** function to pass control to the next middleware function.
3. **Global Middleware**: Middleware mounted using **app.use()** is considered global middleware and is executed for every incoming request to the application. This makes it suitable for tasks that need to be performed on every request, such as setting up common configurations or handling cross-cutting concerns.

Helmet()

**app.use(helmet())** is a middleware function in Express.js that helps secure your web application by setting various HTTP headers to enhance security.

Here's what it does:

1. **Sets Security Headers**: Helmet adds various HTTP headers to your responses to protect your application from certain web vulnerabilities. For example, it sets headers like **X-Content-Type-Options**, **X-Frame-Options**, and **X-XSS-Protection**, among others.
2. **Mitigates Common Attacks**: Helmet helps mitigate common security vulnerabilities such as Cross-Site Scripting (XSS), Cross-Origin Resource Sharing (CORS), Clickjacking, and others by configuring appropriate headers.
3. **Prevents Information Disclosure**: Helmet helps prevent information leakage by setting headers that control browser behavior, such as preventing MIME type sniffing and limiting the amount of information exposed in the **Referrer** header.
4. **Customization**: Helmet allows you to customize its behavior by enabling or disabling specific middleware functions based on your application's needs. You can choose which security headers you want to include and configure them accordingly.

The **helmet.crossOriginResourcePolicy()** middleware in the **helmet** package is used to set the Cross-Origin Resource Policy (CORP) header in HTTP responses. This header is a security feature that allows a server to control which origins are allowed to access its resources across different origins.

Here's what **helmet.crossOriginResourcePolicy()** with the **policy: "cross-origin"** option does:

1. **Cross-Origin Resource Policy Header**: It sets the **Cross-Origin-Resource-Policy** header in HTTP responses to the value **"cross-origin"**. This header specifies that the resource is accessible to cross-origin requests, meaning requests originating from different origins.
2. **Cross-Origin Resource Sharing (CORS)**: The **Cross-Origin-Resource-Policy** header is an alternative to the more widely known **Access-Control-Allow-Origin** header used in Cross-Origin Resource Sharing (CORS). While CORS headers control access to resources from client-side JavaScript code, the **Cross-Origin-Resource-Policy** header provides a server-side mechanism to control access to resources.

The **morgan** middleware in Node.js is used for logging HTTP requests. It automatically logs information about incoming requests to your server, providing insights into the requests being made, including details such as request method, URL, response status, response time, and more.

When you use **app.use(morgan("common"))**, you're instructing the **morgan** middleware to use the "common" predefined format for logging.

Here's what it does:

1. **Logging HTTP Requests**: **morgan("common")** sets up logging for HTTP requests using the "common" log format. This format typically includes the following information:
   * Remote IP address
   * Date and time of the request
   * HTTP method (GET, POST, PUT, etc.)
   * Requested URL
   * HTTP status code
   * Size of the response body

bodyParser

The **bodyParser.json()** middleware in Express.js is used to parse JSON-encoded request bodies. When you use **app.use(bodyParser.json({ limit: "30mb", extended: true }))**, you're configuring the **bodyParser.json()** middleware to handle JSON request bodies with specific options.

Here's what it does:

1. **JSON Parsing**: **bodyParser.json()** is a middleware function that parses incoming request bodies containing JSON data. It automatically parses the JSON data and makes it available on the **req.body** object for further processing by your application's route handlers.
2. **Options Configuration**:
   * **limit: "30mb"**: This option specifies the maximum size of the JSON request body that the middleware will accept. In this case, it's set to **"30mb"**, meaning the middleware will accept JSON bodies up to 30 megabytes in size. If a request body exceeds this limit, the middleware will respond with an HTTP 413 "Payload Too Large" error.
   * **extended: true**: This option allows for parsing of URL-encoded data with rich objects and arrays. Setting it to **true** enables extended syntax, allowing for more complex data structures in the request body.

**bodyParser.urlencoded()**: This middleware is used for parsing URL-encoded data sent in the body of a POST request. URL-encoded data is a format where key-value pairs are encoded in the URL using the **application/x-www-form-urlencoded** content type. The **extended** option in **bodyParser.urlencoded()** determines whether the parsing should use the traditional (default) querystring library (when **extended** is set to **false**) or the qs library (when **extended** is set to **true**).

CORS

Cross-Origin Resource Sharing (CORS) is a security feature implemented by web browsers that controls access to resources (such as APIs or files) on a web page from different origins. An origin is defined by the combination of a scheme (protocol), hostname, and port number.

CORS policies define rules for how a web server should respond to requests from different origins. These policies are enforced by browsers to prevent certain types of attacks, such as Cross-Site Request Forgery (CSRF) and information leakage.

Here's what CORS policies mean:

1. **Same-Origin Policy**: By default, web browsers enforce a same-origin policy, which means that a web page can only make requests to resources on the same origin as the page itself. An origin consists of the protocol (HTTP or HTTPS), domain (hostname), and port number. Any requests made to resources on a different origin are blocked by the browser for security reasons.
2. **Cross-Origin Requests**: CORS policies relax the same-origin policy by allowing web servers to specify which origins are permitted to access their resources. This is done through HTTP headers exchanged between the server and the client. Servers can specify allowed origins, methods, and headers using CORS headers like **Access-Control-Allow-Origin**, **Access-Control-Allow-Methods**, and **Access-Control-Allow-Headers**.
3. **Pre-flight Requests**: For certain types of requests (such as those that use non-simple HTTP methods like PUT or DELETE, or that include custom headers), browsers may send a pre-flight request (an OPTIONS request) to the server to determine if the actual request is allowed. The server responds to this pre-flight request with CORS headers indicating whether the actual request should be permitted.
4. **Granular Control**: CORS policies allow servers to specify granular control over which origins, methods, and headers are allowed. This helps ensure that only trusted origins can access sensitive resources, reducing the risk of unauthorized access and data leakage.

In summary, Cross-Origin Resource Sharing (CORS) policies define rules for how web servers should respond to requests from different origins, allowing servers to specify which origins are permitted to access their resources and providing granular control over the types of requests that are allowed. This helps enhance security by preventing unauthorized access to sensitive resources.

app.use("/assets", express.static(path.join(\_\_dirname, 'public/assets')));

This line of code is setting up a static file server using Express.js. Let's break it down step by step:

1. **app.use()**: This is an Express middleware function used to mount middleware to the Express application. Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle. Here, **app.use()** is being used to mount the static file server middleware.
2. **"/assets"**: This is the base URL path at which the static files will be served. When a client requests a file using a URL that starts with **/assets**, Express will serve the corresponding file from the specified directory. For example, if a client requests **/assets/image.jpg**, Express will serve the **image.jpg** file from the specified directory.
3. **express.static()**: This is a built-in middleware function in Express.js used to serve static files. It takes the root directory from which to serve static assets as an argument and returns middleware that serves static files from that directory.
4. **path.join(\_\_dirname, 'public/assets')**: This constructs an absolute path to the directory containing the static assets. **\_\_dirname** is a global variable in Node.js that represents the directory name of the current module (the module containing this code). **path.join()** is a method from the built-in **path** module in Node.js used to join together path segments. In this case, it joins **\_\_dirname** (the directory containing the current file) with **'public/assets'** to create an absolute path to the directory containing the static assets.

Putting it all together, this line of code sets up a static file server in the Express application. It instructs Express to serve static files located in the **public/assets** directory when requests are made to URLs starting with **/assets**. For example, a request to **/assets/image.jpg** would serve the **image.jpg** file located in the **public/assets** directory. This is commonly used to serve CSS files, JavaScript files, images, and other static assets in web applications

MULTER

Multer used for uploading files, we configure how the uploaded files are stored:  
const storage=multer.diskStorage({

    destination: function(req, file,cb){

        cb(null,"public/assets");

    },

    filename: function(req,file,cb){

        cb(null,file.originalname);

    }

});

This code snippet sets up a disk storage configuration for the multer middleware in a Node.js/Express.js application. Multer is a middleware used for handling multipart/form-data, primarily used for uploading files.

Here's a breakdown of what each part of the configuration does:

1. **const storage = multer.diskStorage({...});**: This initializes disk storage for multer. It creates an instance of multer's DiskStorage engine and assigns it to the variable **storage**. The DiskStorage engine is used to specify how and where uploaded files should be stored on the server's disk.
2. **destination**: This property of the configuration object specifies the directory where uploaded files should be stored. In this case, it's a function that receives three arguments: **req** (the request object), **file** (the file being uploaded), and **cb** (a callback function). Inside the function, **cb(null, "public/assets")** is called, indicating that the files should be stored in the **public/assets** directory. The **cb** function is a callback that must be called once the destination directory is determined. The first argument (**null** in this case) is for errors (if any), and the second argument is the destination directory.
3. **filename**: This property of the configuration object specifies the name of the uploaded file on the server. Similar to **destination**, it's a function that receives the same three arguments (**req**, **file**, and **cb**). Inside the function, **cb(null, file.originalname)** is called, indicating that the uploaded file should retain its original name. The **originalname** property of the **file** object contains the name of the file as it was on the user's computer. Again, the **cb** function is a callback that must be called once the filename is determined.

MONGO DB

In MongoDB, a cluster is not a group of databases or tables; it's a group of servers that work together to store and manage data. Here's a simplified explanation:

1. **MongoDB Database**: In MongoDB, data is organized into databases. Each database can contain multiple collections, which are analogous to tables in relational databases.
2. **Cluster**: A cluster in MongoDB is a group of servers, also known as nodes, that collectively store and manage data. These servers work together to provide fault tolerance, scalability, and high availability for your MongoDB databases.
   * **Replica Set**: The most common type of cluster in MongoDB is a replica set. A replica set consists of multiple MongoDB instances (servers), where one instance serves as the primary and the others are secondary replicas. The primary handles all client requests for reads and writes, while the secondaries replicate data from the primary to provide redundancy and fault tolerance.
   * **Sharded Cluster**: MongoDB also supports sharded clusters for horizontal scaling. In a sharded cluster, data is partitioned across multiple MongoDB instances called shards. Each shard is a replica set that stores a portion of the data. A routing component called **mongos** directs client requests to the appropriate shard based on a sharding key.

In summary, a MongoDB cluster is a group of servers that work together to store and manage data. It's not a group of databases or tables; rather, it's a distributed infrastructure that provides scalability and fault tolerance for MongoDB databases. Within a MongoDB cluster, databases and collections are organized similarly to how they are in traditional databases, but the focus of the cluster is on managing the servers that store and serve the data.

Top of Form

1. **Replica Set**:
   * A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and high availability.
   * In a replica set, there are typically multiple servers, each serving a specific role:
     + Primary: The primary server is the main point of contact for client applications. It receives all write operations and reads data from the local data set. There's only one primary server in a replica set.
     + Secondaries: Secondary servers replicate data from the primary server. They serve read operations and can automatically take over as the primary server in case the current primary fails.
   * Replica sets provide fault tolerance and data redundancy. If the primary server goes down, a secondary can be elected as the new primary, ensuring continuous availability of the database.
2. **Nodes or Servers**:
   * In MongoDB, a node or server refers to an individual instance of the MongoDB database software running on a physical or virtual machine.
   * Each node in a MongoDB cluster can play one of several roles, depending on the configuration:
     + Primary Node: The primary node serves write operations and manages the replica set's configuration.
     + Secondary Node: Secondary nodes replicate data from the primary node and serve read operations. They can also be eligible to become the primary node in case of a failover.
     + Arbiter: An arbiter is a lightweight node that participates in replica set elections but doesn't store any data. Its purpose is to break ties in elections and ensure that there's always an odd number of votes.
   * Nodes communicate with each other to synchronize data and coordinate elections in case of primary node failure.

In summary, a replica set in MongoDB is a group of servers (nodes) that collectively maintain the same data set, providing fault tolerance and high availability. Each node in the replica set has a specific role, such as primary or secondary, and they work together to ensure that data is consistently replicated and available even in the event of failures.

In MongoDB, "network access" refers to the ability of clients, applications, or services to connect to a MongoDB server or cluster over a network and interact with the MongoDB databases hosted on that server or cluster.

In the project, backend runs on port 3001 while frontend works on 3000

PROCESS

n Node.js, **process** is a global object that provides information about the current Node.js process and allows you to interact with it. It is an instance of the **process** module, which is part of the Node.js core API.

Here are some key features and functionalities provided by the **process** object:

1. **Environment Variables**: The **process.env** property is an object containing the user environment. It provides access to environment variables, which are key-value pairs set in the operating system environment. Environment variables can be used to configure and customize the behavior of Node.js applications.
2. **Standard Streams**: The **process.stdin**, **process.stdout**, and **process.stderr** properties represent the standard input, output, and error streams of the Node.js process, respectively. These streams can be used to read input from the user, write output to the console, and output error messages.
3. **Exit Events**: The **process.exit()** method allows you to forcefully terminate the Node.js process. Additionally, the **exit** event is emitted when the Node.js process is about to exit, allowing you to perform cleanup tasks or logging before the process terminates.
4. **Signals Handling**: Node.js processes can listen for operating system signals such as **SIGINT** (generated by pressing **Ctrl+C**), **SIGTERM**, **SIGHUP**, etc. The **process.on()** method allows you to register event listeners for these signals and handle them appropriately.
5. **Memory Usage**: The **process.memoryUsage()** method provides information about the memory usage of the Node.js process, including the amount of memory used, allocated, and available.
6. **CPU Usage**: The **process.cpuUsage()** method provides information about the CPU usage of the Node.js process, including the amount of CPU time used by the user and system.

MONGOOSE SETUP:  
mongoose.connect(process.env.MONGO\_URL,

    {useNewUrlParse: true,

    useUnifiedTopology: true}).then(() => {app.listen(PORT,()=>console.log(`Server Port: ${PORT}`))}).catch(error => { console.log(`Server error: ${error}

1. **mongoose.connect(process.env.MONGO\_URL, {...})**: This initiates an asynchronous operation to connect to a MongoDB database using Mongoose. It uses the URL provided in the environment variable **MONGO\_URL** and applies options like **useNewUrlParser** and **useUnifiedTopology**.
2. **.then(() => { app.listen(PORT, () => console.log(**Server Port: ${PORT}**)) })**: Once the connection to the MongoDB database is successfully established, the promise returned by **mongoose.connect()** is fulfilled, and the **then()** method is invoked. Within the **then()** callback, it starts the Express.js server to listen on the specified port (**PORT**) by calling **app.listen()**. Upon successful server initialization, it logs a message indicating the server is running and listening on that port.
3. **.catch(error => { console.log(**Server error: ${error}**)})**: Additionally, if an error occurs during the execution of the **mongoose.connect()** function or the subsequent **app.listen()** function, the **catch()** method will be invoked. It allows you to handle any errors that may occur during the connection or server startup process. In this case, it simply logs the error message to the console.

The **app.listen()** function is an Express.js method used to start a server that will listen for incoming HTTP requests on a specified port. It's typically the last step in setting up an Express.js server.

The function after PORT in the then() is the callback function