Important functions

What is `fetch()`?

> `fetch()` is a built-in JavaScript function used to make HTTP requests (like GET, POST, PUT, DELETE) to servers or APIs — just like how your browser loads web pages.

It returns a Promise, which resolves to a Response object containing the data from the server.

Basic Syntax:

```
fetch(url, options)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Error:", error));
```

Parameters:

1. **`url`** → The endpoint where you're sending the request.
   Example: `"http://localhost:3000/api/users"`

2. **`options`** (optional) → An object that configures:
   - HTTP **method** (`GET`, `POST`, etc.)
   - **headers** (like `"Content-Type": "application/json"`)
   - **body** (for sending data in JSON format)

db.js

Line	Code / Concept
`const mongoose = require("mongoose");`	Imports the Mongoose library to connect and interact with MongoDB.
`const connectDB = async () => { ... }`	Defines an asynchronous function (since database operations take time).
`await mongoose.connect(URI)`	Connects to the database using a connection string stored in an environment variable named `MONGO_URI`.
`console.log("MongoDB connected successfully!");`	Prints a success message if the connection works.
`catch (err) { console.log("DB connection failed:", err.message); }`	Catches and logs any connection error, making debugging easier.
`module.exports = connectDB;`	Exports the function so it can be reused in `server.js` or `app.js`.

userModel.js / articleModel.js

Concept	Description
Import Mongoose	Mongoose is the ODM (Object Data Modeling) library used to define schemas and interact with MongoDB.

`mongoose.Schema`	A blueprint that defines how documents (records) in a MongoDB collection are structured.
`mongoose.model()`	Converts the schema into a usable model that represents a MongoDB collection and allows performing CRUD operations.
Module Export	Exports the created model so it can be imported and used across the application (e.g., in routes or controllers).

userController.js

Function / Method	Category	Explanation / Purpose	Example Scenario
`require()`	Node.js built-in	Imports external modules (like `userModel`) so you can use them inside this file.	Import the user model file to access database operations.
`async` / `await`	JavaScript (Async Handling)	Used for handling asynchronous operations like database queries without blocking the main thread.	Waits for MongoDB to finish creating or finding a user before continuing.

`try { } catch (error) { }`	Error Handling	Ensures the app doesn't crash if something fails. The error is caught and logged or returned with a proper message.	Prevents the app from breaking when database connection fails.
`userModel.create()`	Mongoose Method	Inserts a new document (record) into the `users` collection based on the schema.	Used in both `createAdmin()` and `createUser()` to add users.
`userModel.findOne()`	Mongoose Method	Searches for a single document that matches specific conditions. Returns `null` if not found.	Used in `getUserById()` to find a user by name and password.
`console.error()` / `console.log()`	Node.js Logging	Displays messages in the terminal — useful for debugging.	Logs "User created successfully" or shows an error message.
`res.status()`	Express Response	Sets the HTTP response status code (like 200, 201, 404, 500).	`res.status(404)` is used when a user is not found.
`res.json()`	Express Response	Sends a JSON response to the client. Usually follows `res.status()`.	Returns `{ message: "User created successfully"`

			`}` after adding a user.
`module.exports`	Node.js Export	Exports variables or functions so other files (like routes) can use them.	Exports `{ createAdmin, createUser, getUserById }` for reuse.

userRoutes.js

Function / Method	Category	Explanation / Purpose	Example Scenario
`require()`	Node.js Module System	Imports external modules like `express` or controller functions so they can be used in this file.	Used to load `express` and functions from `userController`.
`express.Router()`	Express Method	Creates a **new router object** to define and group related routes. Keeps route logic modular and clean.	Used to create a router for user-related endpoints.
`router.post()`	Express Routing	Defines a **POST route** that listens for POST requests at a specific path (e.g., `/createUser`). Executes the given	Handles user creation or login submissions sent from frontend forms.

		controller function when triggered.	
`module.exports`	Node.js Export	Exports the router so it can be imported into the main app (like `server.js`).	Enables the main application to use this router's endpoints.

articleController.js

Function / Method	Category	Explanation / Purpose	Example Scenario
`require()`	Node.js Module System	Imports external modules or files like the `articleModel` so you can interact with the MongoDB collection.	Loads the article model from the models folder.

`async` / `await`	JavaScript (Asynchronous Handling)	Manages database operations that take time to complete without freezing the server.	Waits for `articleModel.create()` to finish before sending a response.
`try { } catch (error) { }`	Error Handling	Ensures the app continues running even if a database or logic error occurs.	Logs errors like failed article creation or invalid IDs.
`articleModel.create()`	Mongoose Method	Inserts a new document (article) into the MongoDB collection.	Used when creating a new article via API.
`articleModel.find()`	Mongoose Method	Retrieves all documents from the collection.	Used to list all available articles.

`articleModel.findByIdAndUpdate()`	Mongoose Method	Finds an article by its unique ID and updates specific fields.	Used to edit an existing article's title or description.
`articleModel.findByIdAndDelete()`	Mongoose Method	Finds and deletes an article by its ID.	Used to remove an article permanently.
`res.status()`	Express Response	Sets an HTTP status code for the response (like 200, 201, 404, 500).	`res.status(201)` means "Created successfully."
`res.json()`	Express Response	Sends a structured JSON object as a response to the client.	Returns `{ message: "Article updated successfully" }` after updating.
`req.body`	Express Request	Captures data sent in the body of a POST or PUT request.	Used when the frontend sends article data to create or update.

`req.params`	Express Request	Captures route parameters (like `id` from `/article/:id`).	Used to find or delete an article by ID.
`console.error()` / `console.log()`	Node.js Logging	Displays messages and errors in the terminal for debugging.	Helps track issues during database operations.
`module.exports`	Node.js Export	Exports the controller functions so routes can import and use them.	Exports `{ createArticle, getAllArticles, updateArticle, deleteArticle }`.

Pragmatics for the above func:

Concept	Example (Conceptual)	Explanation
Create Document	`articleModel.create(req.body)`	Adds a new article to MongoDB.

Read All Documents	`articleModel.find()`	Retrieves every article.
Update Document	`articleModel.findByIdAndUpdate(id, data)`	Updates selected fields of an article.
Delete Document	`articleModel.findByIdAndDelete(id)`	Removes an article permanently.
Respond to Client	`res.status(200).json({ message })`	Sends success/failure message to frontend.

articleRoutes.js

Function / Concept	Type	Description
`require("express")`	Import	Imports the Express framework.
`express.Router()`	Function	Creates a new router object to define routes separately from the main app.
`router.post(path, handler)`	Method	Defines a POST route (used here for creating a new article).
`router.get(path, handler)`	Method	Defines a GET route (used here to fetch all articles).
`router.put(path, handler)`	Method	Defines a PUT route (used here to update an existing article by `id`).

`router.delete(path, handler)`	Method	Defines a DELETE route (used here to remove an article by `id`).
`module.exports = router`	Export	Exports the router object to be used in the main server file (`app.js`).
`{ createArticle, getAllArticles, updateArticle, deleteArticle }`	Destructuring Import	Imports controller functions from `articleController.js` that handle business logic for each route.

server.js

Function / Concept	Type	Description
`require("express")`	Import	Imports the Express framework for creating the HTTP server.
`require("dotenv")`	Import	Loads environment variables from a `.env` file into `process.env`.
`dotenv.config()`	Function	Activates dotenv so you can access environment variables.
`connectDB()`	Function	Connects to MongoDB using Mongoose (from your config file).
`express()`	Function	Creates an Express application instance.

`bodyParser.json()`	Middleware	Parses incoming requests with JSON payloads.
`bodyParser.urlencoded({ extended: true })`	Middleware	Parses URL-encoded request bodies (for form submissions).
`cors()`	Middleware	Enables Cross-Origin Resource Sharing (CORS) — allows frontend apps from different domains to access your API.
`app.use(path, router)`	Middleware	Mounts route files (like `userRoutes` or `articleRoutes`) under specific API prefixes.
`app.get("/", callback)`	Route	Defines a simple GET route to respond to the root endpoint.
`app.listen(PORT, callback)`	Method	Starts the server and listens on the specified port.
`process.env.PORT`	Environment Variable	Reads the port number from the `.env` file; defaults to `3000` if not provided.
`createAdmin()`	Custom Function	Creates an admin user (called once if no admin exists).

Script.js

Function / Concept	Type	Description
`document.getElementById()`	DOM Method	Used to access HTML elements (form, title, buttons) for dynamic interaction.
`isSignup`	Boolean Flag	Keeps track of whether the form is in **signup** or **login** mode.
`backendURL`	Constant	Stores the base API endpoint to connect frontend with backend (`http://localhost:3000/api/users/`).
`toggleForm.addEventListener("click", …)`	Event Listener	Toggles the form UI between *signup* and *login* modes by changing labels and button text dynamically.
`form.addEventListener("submit", async …)`	Event Listener	Handles form submission for both signup and login using `fetch`.
`preventDefault()`	DOM Method	Stops the form from refreshing the page upon submit.
`value.trim()`	String Method	Cleans up user input by removing unwanted spaces.

Validation Check	Conditional	Ensures both name and password fields are filled before sending a request.
Dynamic Endpoint Selection	Conditional Logic	Chooses between `createUser` (for signup) and `findUser` (for login) based on `isSignup`.
`fetch()`	Web API Function	Sends HTTP requests to the backend. Uses `POST` with headers and JSON body.
`"Content-Type": "application/json"`	Header	Tells the server the body content is JSON format.
`JSON.stringify()`	Function	Converts JavaScript object (form data) into JSON string for sending to backend.
`await res.json()`	Async Function	Converts the server's JSON response back into a JavaScript object.
`res.ok` Check	Conditional	Checks if the response was successful (HTTP 200-299).
`localStorage.setItem()`	Web Storage API	Stores the user's `role` (like `user` or `admin`) in browser storage for later use.
`window.location.href`	Browser Property	Redirects to another page (e.g., `article.html`) after successful login/signup.

`alert()`	UI Function	Displays quick feedback messages for success or error cases.
`try...catch`	Error Handling	Catches and displays network or server-side errors.

articleController.js

Function / Concept	Type	Description
`localStorage.getItem("role")`	Web Storage API	Retrieves the saved user role (`user` or `admin`) from local storage to control UI visibility.
`document.getElementById()`	DOM Method	Fetches references to form fields, buttons, containers, and templates for dynamic manipulation.
`form.hidden = role !== "admin"`	DOM Property	Hides the create-article form unless the logged-in user is an admin.
`textContent`	DOM Property	Updates UI text dynamically (like showing current role).

⚙️ Helper Function — `apiRequest()`

Part	Explanation
`async function apiRequest(path, method, body)`	A reusable wrapper for `fetch()` to handle GET, POST, PUT, DELETE easily.
`options.headers["Content-Type"] = "application/json"`	Tells the backend that the body data is JSON.
`if (body) options.body = JSON.stringify(body)`	Converts JS object to JSON string before sending.
`await fetch(API_URL + path, options)`	Makes the network request to the backend API.
`response.json()`	Converts the server's JSON response into a JS object.
Purpose	Simplifies all HTTP calls across the app (DRY principle).

`renderArticle(article)`

| Purpose | Dynamically displays each article card using a `<template>` tag. |
| Key Methods |

- `template.content.cloneNode(true)` → clones hidden HTML template

- `card.dataset.id` → attaches the article ID for edit/delete actions
- `actions.hidden = false` → shows edit/delete buttons if user is admin |
  | Usage | Called for each article fetched from backend. |

`loadArticles()`

| Action | Fetches all articles from backend and renders them. |
| Steps |

1. Clears existing content: `container.innerHTML = ""`
2. Calls `apiRequest()` (GET)
3. Iterates over result and calls `renderArticle(article)` for each. |

🧑‍💻 `handleAdminAction(e)`

| Purpose | Handles **Edit**, **Save**, and **Delete** actions for admins. |
| Logic |

Step	Description
`e.target.closest(".card")`	Finds which card was clicked.
`if (role !== "admin") return;`	Prevents non-admin users from editing/deleting.

Edit	Makes title and description editable, toggles visibility of buttons.
Save	Sends PUT request with updated content via `apiRequest("/id", "PUT", {...})`.
Delete	Asks for confirmation and sends DELETE request to remove article.

`createArticle()`

| Purpose | Allows admin to create new articles via form. |
| Steps |

1. Reads title & description inputs.
2. Validates both fields (alert if empty).
3. Sends POST request using `apiRequest("", "POST", { title, description })`.
4. Clears inputs and reloads all articles (`loadArticles()`). |

Concept	Explanation	Example
Template Cloning	Efficiently renders multiple cards using one HTML template.	`<template>` → clone → customize → append
Dynamic Role-Based UI	Restricts article creation/editing to admin users only.	`form.hidden = role !== "admin"`

Reusable Fetch Wrapper	Centralizes all API calls into one helper for clarity.	`apiRequest("/id", "PUT", {...})`
LocalStorage Role Handling	Persists user role across pages.	Retrieved from login/signup page.