FSD INTERNAL ALL ANSWERS

1. (a) ExpressJS: Routes and Parameters

🎯 Aim

To write an ExpressJS program that defines a basic route, a route with parameters, and handles query parameters.

📚 Description

Express allows us to map URLs (routes) to specific handler functions.

- **Route:** A simple path like `/` or `/about`.
- **Route Parameters:** Dynamic parts of a URL, useful for unique items like a user ID. We define them with a colon (e.g., `/user/:id`). Express makes the value available in `req.params.id`.
- **Query Parameters:** Key-value pairs at the end of a URL after a `?`, used for filtering or searching (e.g., `/search?q=test`). Express makes these available in `req.query.q`.

💻 Program (`index.js`)

JavaScript

```
const express = require('express');
const app = express();
const port = 3000;

// 1. Basic Route
app.get('/', (req, res) => {
  res.send('Hello! This is the homepage.');
});

// 2. Route with Parameters
// Try visiting: http://localhost:3000/user/123
app.get('/user/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`You are viewing the profile for User ID: ${userId}`);
});

// 3. Route with Query Parameters
// Try visiting: http://localhost:3000/search?q=react&lang=js
app.get('/search', (req, res) => {
  const searchQuery = req.query.q;
  const langQuery = req.query.lang;
  res.send(`You searched for: "${searchQuery}" in language: "${langQuery}"`);
});
```

```
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

🚀 Execution Steps

1. Create a folder (e.g., `lab-1a`).
2. Inside, run `npm init -y` to create `package.json`.
3. Install Express: `npm install express`.
4. Create a file named `index.js` and paste the code above.
5. Run the server: `node index.js`.
6. Open your browser and test the different URLs mentioned in the code comments.

📊 Sample Output

Browser (visiting `http://localhost:3000/`):
Hello! This is the homepage.

-

Browser (visiting `http://localhost:3000/user/123`):
You are viewing the profile for User ID: 123

-

Browser (visiting `http://localhost:3000/search?q=react&lang=js`):
You searched for: "react" in language: "js"

-

💡 Memory Tips

- **P**arams = **P**ath (part of the URL path) -> `req.params`
- **Q**uery = **Q**uestion mark (after the `?`) -> `req.query`
- `req` = Request (incoming data).
- `res` = Response (outgoing data).

---

1. (b) ReactJS: Sharing Data Between Components

🎯 Aim

To write a ReactJS program demonstrating how to share data from a parent component to a child component (and back up).

📚 Description

Data in React flows in one direction: **down** (from parent to child) via **props**.

1. **Parent to Child:** The parent component passes data as attributes (props) to the child component.
2. **Child to Parent:** To send data *up*, the parent passes a *function* (that updates its own state) as a prop to the child. The child then calls this function when an event occurs.

This example shows a `Parent` component holding a message. It passes the message *down* to the `Child` and also passes a function *down* to let the `Child` change the message.

💻 Program (`App.js` and `Child.js`)

`src/App.js` (Parent Component)

JavaScript

```
import React, { useState } from 'react';
import Child from './Child'; // Import the child
import './App.css';

function App() {
  const [message, setMessage] = useState("Hello from the Parent!");

  // Function to be passed to the child
  const updateMessageFromChild = (newMessage) => {
    setMessage(newMessage);
  };

  return (
    <div className="App">
      <h1>Parent Component</h1>
      <p>Message: {message}</p>

      {/* Pass data DOWN (message)
        Pass function DOWN (updateMessageFromChild)
      */}
      <Child
        parentMessage={message}
        onUpdateMessage={updateMessageFromChild}
      />
    </div>
  );
}

export default App;
```

`src/Child.js` (Child Component)

JavaScript
```
import React from 'react';

// Receive props from the parent
function Child(props) {

  const handleUpdate = () => {
    // Call the function passed from the parent
    props.onUpdateMessage("Message changed by the Child!");
  };

  return (
    <div className="child-box">
      <h2>Child Component</h2>
      {/* Display the prop passed from the parent */}
      <p>Message from parent: {props.parentMessage}</p>

      {/* Button to send data "up" by calling the prop function */}
      <button onClick={handleUpdate}>Change Parent's Message</button>
    </div>
  );
}

export default Child;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. Navigate into the app: `cd my-app`.
3. Replace the code in `src/App.js` with the parent component code.
4. Create a new file `src/Child.js` and paste the child component code.
5. (Optional) Add a border to `src/App.css`: `.child-box { border: 1px solid #ccc; padding: 10px; margin-top: 20px; }`.
6. Start the app: `npm start`.

📊 Sample Output

Initially:

> **Parent Component** Message: Hello from the Parent!
>
> **Child Component** Message from parent: Hello from the Parent! [Change Parent's Message]

After clicking the button:

> **Parent Component** Message: Message changed by the Child!
>
> **Child Component** Message from parent: Message changed by the Child!
> [Change Parent's Message]

💡 Memory Tips

- **Data Down:** Pass data as a prop: `<Child myData={data} />`.
- **Data Up:** Pass a *function* as a prop: `<Child onUpdate={handleUpdate} />`.

---

2. (a) ReactJS: Props and States

🎯 Aim

To write a React JS program that clearly differentiates between `props` and `state`.

📚 Description

- **Props (Properties):** These are read-only values passed *into* a component from its parent. A component **cannot** change its own props. They are like arguments passed to a function.
- **State:** This is data managed *inside* a component. It is private to that component and can be changed over time (using the `useState` hook). When state changes, React re-renders the component.

This program has a parent `App` that holds a `count` in its **state**. It passes this `count` as a **prop** to a child `CounterDisplay` component.

💻 Program (`App.js` and `CounterDisplay.js`)

`src/App.js` (Parent)

JavaScript
```
import React, { useState } from 'react';
import CounterDisplay from './CounterDisplay';
import './App.css';

function App() {
  // 'count' is STATE managed by App
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1); // Changing the state
  };
```

```
  return (
    <div className="App">
      <h1>Parent Component (Manages State)</h1>
      <p>The current count is: {count}</p>
      <button onClick={incrementCount}>Increment Count</button>

      <hr />

      {/* 'countValue' is a PROP being passed to the child */}
      <CounterDisplay countValue={count} />
    </div>
  );
}

export default App;
```

`src/CounterDisplay.js` (Child)

```
JavaScript
import React from 'react';

// 'props' object contains all props passed from the parent
function CounterDisplay(props) {
  // This component receives 'countValue' as a prop
  // It is read-only.

  // This would cause an error:
  // props.countValue = 10; // !! ERROR: props are read-only !!

  return (
    <div>
      <h2>Child Component (Receives Props)</h2>
      <p>The parent's count is: <strong>{props.countValue}</strong></p>
    </div>
  );
}

export default CounterDisplay;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. `cd my-app`.
3. Replace `src/App.js` with the code above.

4. Create `src/CounterDisplay.js` and paste the code above.
5. Start the app: `npm start`.

📊 Sample Output

The browser will display:

> **Parent Component (Manages State)** The current count is: 0 [Increment Count]
>
> ---
>
> **Child Component (Receives Props)** The parent's count is: **0**

When you click the `[Increment Count]` button, the state in `App` changes. This causes `App` to re-render, and it passes the new count (e.g., 1) as a prop to `CounterDisplay`, which also re-renders.

> **Parent Component (Manages State)** The current count is: 1 [Increment Count]
>
> ---
>
> **Child Component (Receives Props)** The parent's count is: **1**

💡 Memory Tips

- **S**tate = **S**tays (inside the component, changeable).
- **P**rops = **P**assed (from parent, read-only).
- Use `useState` to manage `state`.
- Access `props` via the `props` argument.

---

2. (b) ReactJS: Add Styles (CSS & Sass)

🎯 Aim

To demonstrate three ways to add styles to a React JS component: CSS Stylesheets, Inline Styles, and Sass.

📚 Description

1. **CSS Stylesheets:** The standard method. You import a `.css` file and use the `className` attribute in JSX (instead of `class`).
2. **Inline Styles:** Used for dynamic or specific styles. You pass a JavaScript *object* to the `style` attribute. Property names must be `camelCase` (e.g., `fontSize` instead of `font-size`).

3. **Sass/SCSS:** A CSS preprocessor that adds features like variables, nesting, and mixins. You need to install the `sass` package, and then you can import `.scss` files just like `.css` files.

💻 Program

`src/App.css` (Standard CSS File)

```
CSS
/* This is a standard CSS file */
.box {
  border: 1px solid blue;
  padding: 10px;
  margin: 10px;
}

.box-title {
  color: blue;
  font-weight: bold;
}
```

`src/styles.scss` (Sass File)

```
SCSS
/* This is a Sass file */
$primary-color: #c0392b; // Sass variable

.sass-box {
  border: 1px solid $primary-color;
  padding: 10px;
  margin: 10px;

  /* Sass nesting */
  .sass-box-title {
    color: $primary-color;
    font-weight: bold;
  }
}
```

`src/App.js` (Component)

```
JavaScript
import React from 'react';

// 1. Importing the CSS stylesheet
import './App.css';
```

```
// 2. Importing the SASS stylesheet
import './styles.scss';

function App() {

  // 3. Defining an object for Inline Styles
  const inlineStyleObject = {
    border: '1px solid green',
    padding: '10px',
    margin: '10px',
  };

  const inlineTitleStyle = {
    color: 'green',
    fontWeight: 'bold',
  };

  return (
    <div>
      <h1>Styling in React</h1>

      {/* 1. Using CSS Stylesheet */}
      <div className="box">
        <p className="box-title">Styled with App.css</p>
      </div>

      {/* 2. Using Sass/SCSS Stylesheet */}
      <div className="sass-box">
        <p className="sass-box-title">Styled with styles.scss</p>
      </div>

      {/* 3. Using Inline Styles */}
      <div style={inlineStyleObject}>
        <p style={inlineTitleStyle}>Styled with Inline Style Object</p>
      </div>

    </div>
  );
}

export default App;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.

2. `cd my-app`.
3. **Install Sass:** `npm install sass`.
4. Replace `src/App.js` with the code above.
5. Replace `src/App.css` with the code above.
6. Create a new file `src/styles.scss` and paste the code above.
7. Start the app: `npm start`.

📊 Sample Output

The browser will show three boxes:

1. A box with a **blue** border and **blue** text.
2. A box with a **red** (`#c0392b`) border and **red** text.
3. A box with a **green** border and **green** text.

💡 Memory Tips

- HTML `class` = JSX `className`.
- HTML `style="color:red;"` = JSX `style={{ color: 'red' }}` (double curly braces: one for JSX, one for the object).
- For Sass, just `npm install sass` and `import './styles.scss'`.

---

3. (a) ExpressJS: Templating Engine (EJS)

🎯 Aim

To write an ExpressJS program that uses the EJS (Embedded JavaScript) templating engine to render dynamic HTML pages.

📚 Description

A templating engine lets us create dynamic HTML on the server. We write an HTML template (like `index.ejs`) and use special tags to insert data (variables, loops) from our Express server. Express then *renders* this template into a plain HTML string and sends it to the user. EJS is popular because it looks just like HTML with a few extra tags.

💻 Program

`index.js` (Server File)

JavaScript
```
const express = require('express');
const app = express();
const port = 3000;
```

```
// 1. Set EJS as the view engine
app.set('view engine', 'ejs');

// By default, Express looks for views in a folder named "views"
// We don't need to set this, but it's good to know.
// const path = require('path');
// app.set('views', path.join(__dirname, 'views'));

// 2. Create a route to render a view
app.get('/', (req, res) => {
  // Data to pass to the template
  const data = {
    title: 'Home Page',
    message: 'Hello from EJS!',
    items: ['Apple', 'Banana', 'Cherry']
  };

  // 3. Render 'index.ejs' and pass the data object
  res.render('index', data);
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

`views/index.ejs` (Template File)

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title><%= title %></title>
  <style>
    body { font-family: sans-serif; padding: 20px; }
    li { color: navy; }
  </style>
</head>
<body>

  <h1><%= message %></h1>

  <p>Here is a list of items:</p>
  <ul>
    <% items.forEach(function(item) { %>
      <li><%= item %></li>
```

```
  <% }); %>
 </ul>

</body>
</html>
```

🚀 Execution Steps

1. Create a folder (e.g., `lab-3a`).
2. `npm init -y`.
3. Install dependencies: `npm install express ejs`.
4. Create `index.js` and paste the server code.
5. **Create a folder named `views`.**
6. Inside the `views` folder, create `index.ejs` and paste the template code.
7. Run the server: `node index.js`.
8. Visit `http://localhost:3000` in your browser.

📊 Sample Output

The browser will show a fully rendered HTML page:

> **Home Page**
>
> **Hello from EJS!**
>
> Here is a list of items:
>
> - Apple
> - Banana
> - Cherry

💡 Memory Tips

- `app.set('view engine', 'ejs')` = Tell Express to use EJS.
- `res.render('filename', { data })` = Render the file, pass the data.
- `<%= variable %>` = **Print** the variable (e.g., `<p><%= name %></p>`).
- `<% logic %>` = **Run** the logic (e.g., `<% if (user) { %>`).

---

3. (b) ReactJS: Render HTML to a Web Page

🎯 Aim

To write the most basic React JS program that renders a piece of HTML (JSX) to the web page.

📚 Description

This is the core of how a React application starts.

1. We have a single HTML file (`index.html`) which is the "shell." It contains a single `<div>` (usually with `id="root"`).
2. Our JavaScript (`index.js`) uses the `ReactDOM` library to "take over" this root div.
3. We tell `ReactDOM` to render our main component (usually `<App />`) *inside* that div.
4. The `<App />` component (and all its children) then controls everything the user sees.

This process uses the "Virtual DOM." React renders our components in memory, then `ReactDOM` efficiently updates the *real* DOM (the `div#root`) to match.

💻 Program

This program uses the standard `create-react-app` file structure.

`public/index.html` (The HTML Shell)

HTML
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>React App</title>
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>

</body>
</html>

`src/App.js` (The Component to be Rendered)

JavaScript
import React from 'react';

// This is a simple React component that returns JSX (which looks like HTML)
function App() {
  return (

      <h1>Hello, React!</h1>
      This HTML is being rendered by a React component.

  );

```
}

export default App;
```

`src/index.js` (The "Glue")

```
JavaScript
import React from 'react';
import ReactDOM from 'react-dom/client'; // Import the new client for React 18
import App from './App'; // Import our component

// 1. Get the root element from index.html
const rootElement = document.getElementById('root');

// 2. Create a "root" for React to manage
const root = ReactDOM.createRoot(rootElement);

// 3. Tell React to render our <App /> component inside the root
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. `cd my-app`.
3. The files above are already created for you by `create-react-app`. You can inspect them to see how they work.
4. Start the app: `npm start`.

📊 Sample Output

The browser will show:

> **Hello, React!**
>
> This HTML is being rendered by a React component.

💡 Memory Tips

- `index.html` has the "target" (`<div id="root">`).
- `index.js` is the "launcher."

- `ReactDOM.createRoot(target).render(<Component />)` is the launch command.
- `App.js` is the main component that gets launched.

---

4. (a) ExpressJS: Middleware

🎯 Aim

To write an ExpressJS program that demonstrates the use of middleware.

📚 Description

Middleware functions are the "gatekeepers" of Express. They are functions that run *between* the incoming request and the final route handler. They have access to the request (`req`), response (`res`), and a special function called `next`.

Middleware can:

- Execute any code.
- Modify the `req` or `res` objects.
- End the request-response cycle (by sending a response).
- Call the *next* middleware in the stack (by calling `next()`).

This program creates a simple "logger" middleware that logs the time, method, and URL of *every* request that comes into the server.

💻 Program (`index.js`)

JavaScript

```
const express = require('express');
const app = express();
const port = 3000;

// 1. Define a middleware function
const loggerMiddleware = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);

  // 2. CRITICAL: Call next() to pass control to the next function
  // If you forget this, the request will hang forever!
  next();
};

// 3. Use the middleware (apply it to ALL routes)
app.use(loggerMiddleware);

// --- Define Routes ---
```

```
app.get('/', (req, res) => {
  res.send('Homepage. Check your terminal to see the log!');
});

app.get('/about', (req, res) => {
  res.send('About Page. Check your terminal!');
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

🚀 Execution Steps

1. Create a folder (e.g., `lab-4a`).
2. `npm init -y`.
3. Install Express: `npm install express`.
4. Create `index.js` and paste the code.
5. Run the server: `node index.js`.
6. In your browser, visit `http://localhost:3000/`.
7. Then, visit `http://localhost:3000/about`.

📊 Sample Output

Browser (visiting `/`):

Homepage. Check your terminal to see the log!

Browser (visiting `/about`):

About Page. Check your terminal!

Node.js Terminal (where `node index.js` is running):

```
Server running at http://localhost:3000
[2025-11-08T09:00:01.000Z] GET /
[2025-11-08T09:00:01.005Z] GET /favicon.ico
[2025-11-08T09:00:05.000Z] GET /about
[2025-11-08T09:00:05.005Z] GET /favicon.ico
```

(The middleware successfully logged every request before the route handler sent a response.)

💡 Memory Tips

- Middleware is the "man in the middle."
- It has three arguments: `(req, res, next)`.
- **Must** call `next()` to continue, or `res.send()` to stop.
- `app.use(myMiddleware)` = Apply to *all* routes.

---

4. (b) ReactJS: Responding to Events

🎯 Aim

To write a React JS program that responds to user events, such as button clicks and input changes.

📚 Description

React components can respond to user actions using **event handlers**. The syntax is similar to HTML but follows `camelCase` conventions.

- HTML: `onclick` -> React: `onClick`
- HTML: `onchange` -> React: `onChange`

We pass a *function* to the event handler prop. This function will be called whenever the event occurs.

💻 Program (`App.js`)

JavaScript

```
import React, { useState } from 'react';
import './App.css';

function App() {
  // State to hold the value of the input
  const [inputValue, setInputValue] = useState("");

  // 1. Event handler for a button click
  const handleButtonClick = () => {
    alert('Button was clicked!');
  };

  // 2. Event handler for an input change
  const handleInputChange = (event) => {
    // 'event.target.value' holds the current value of the input
    setInputValue(event.target.value);
  };
```

```
  // 3. Event handler for a form submission
  const handleSubmit = (event) => {
    event.preventDefault(); // Prevents the page from reloading
    alert(`Form submitted with value: ${inputValue}`);
  };

  return (
    <div className="App">
      <h1>Event Handling</h1>

      {/* 1. onClick event */}
      <button onClick={handleButtonClick}>Click Me!</button>

      <hr />

      {/* 3. onSubmit event */}
      <form onSubmit={handleSubmit}>
        <p>You are typing: <strong>{inputValue}</strong></p>

        {/* 2. onChange event */}
        <input
          type="text"
          value={inputValue}
          onChange={handleInputChange}
          placeholder="Type something..."
        />
        <button type="submit">Submit</button>
      </form>
    </div>
  );
}

export default App;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. `cd my-app`.
3. Replace `src/App.js` with the code above.
4. Start the app: `npm start`.

📊 Sample Output

The browser shows a button and a form.

- **Clicking the "Click Me!" button:** An alert box appears saying "Button was clicked!".

- **Typing in the input box:** The text "You are typing: ..." updates in real-time with what you type.
- **Clicking the "Submit" button:** An alert box appears saying "Form submitted with value: [your text]". The page does *not* reload.

💡 Memory Tips

- Event: `onEventName` (e.g., `onClick`, `onChange`).
- Handler: `{functionName}` (e.g., `onClick={handleClick}`).
- **Important:** Pass the function *reference* (`{handleClick}`), not the *result* (`{handleClick()}`). Using `()` would call the function immediately on render.

---

5. (a) ExpressJS: User Authentication (Simple Session)

🎯 Aim

To write a basic ExpressJS program for user authentication using sessions.

📚 Description

This program demonstrates a "protected" route.

1. We use `express-session` to give each user a unique "session" (like a temporary ID card).
2. A user `POST`s their credentials to `/login`.
3. If valid, we store their user info in `req.session.user`. This is like "signing" their ID card.
4. We create a middleware `checkAuth` to protect routes. This middleware *checks* if `req.session.user` exists.
5. The `/profile` route is protected. If the user is logged in (session exists), they see it. If not, they are redirected to `/login`.
6. The `/logout` route destroys the session, "logging them out."

💻 Program (`index.js`)

JavaScript
```
const express = require('express');
const session = require('express-session');
const app = express();
const port = 3000;

// --- Middleware ---
// To parse POST data from forms
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

```
// 1. Session Middleware
app.use(session({
  secret: 'a-very-secret-key-12345', // Change this in production!
  resave: false,
  saveUninitialized: true,
}));

// 2. Auth Checking Middleware
const checkAuth = (req, res, next) => {
  if (req.session.user) {
    // User is logged in, proceed
    next();
  } else {
    // User is not logged in, redirect
    res.redirect('/login');
  }
};

// --- Routes ---

// Public route - shows a login form
app.get('/login', (req, res) => {
  res.send(`
    <h1>Login</h1>
    <form action="/login" method="POST">
      <input name="username" placeholder="Username (use 'admin')" /><br />
      <input name="password" type="password" placeholder="Password (use 'pass')" /><br />
      <button type="submit">Login</button>
    </form>
  `);
});

// 3. Login processing
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // In a real app, you'd check a database. Here we hardcode.
  if (username === 'admin' && password === 'pass') {
    // 4. Save user to session
    req.session.user = { username: 'admin', role: 'admin' };
    res.redirect('/profile');
  } else {
    res.send('Invalid username or password');
  }
});

// 5. Protected Route
```

```
app.get('/profile', checkAuth, (req, res) => {
  // This route only runs if checkAuth() calls next()
  res.send(`
    <h1>Welcome, ${req.session.user.username}!</h1>
    <p>This is your protected profile.</p>
    <a href="/logout">Logout</a>
  `);
});

// 6. Logout Route
app.get('/logout', (req, res) => {
  req.session.destroy(err => {
    if (err) return res.send('Error logging out');
    res.redirect('/login');
  });
});

app.get('/', (req, res) => res.redirect('/login'));

app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

🚀 Execution Steps

1. Create a folder (e.g., `lab-5a`).
2. `npm init -y`.
3. Install dependencies: `npm install express express-session`.
4. Create `index.js` and paste the code.
5. Run the server: `node index.js`.

📊 Sample Output

1. Visit `http://localhost:3000/profile`. You are *not* logged in, so you are redirected to `/login`.
2. The `/login` page shows the form.
3. Enter "admin" and "pass" and click "Login."
4. You are redirected to `/profile`, which now shows: "Welcome, admin!".
5. Click "Logout." You are redirected back to `/login`.
6. If you try visiting `/profile` again, it will redirect you to `/login`.

💡 Memory Tips

- **Session:** A "backpack" (`req.session`) for the user.
- **Login:** Puts user info *into* the backpack: `req.session.user = ...`.
- **Protect:** *Checks* the backpack: `if (req.session.user)`.
- **Logout:** *Empties* the backpack: `req.session.destroy()`.

---

5. (b) MongoDB: CRUD Operations

🎯 Aim

To write MongoDB queries in the `mongosh` shell for all CRUD (Create, Read, Update, Delete) operations.

📚 Description

These are the four fundamental operations for any database.

- **C**reate: Add new documents (rows) to a collection (table).
- **R**ead: Find and retrieve existing documents.
- **U**pdate: Modify existing documents.
- **D**elete: Remove existing documents.

The queries below are for the `mongosh` terminal.

💻 Program (Queries)

JavaScript

```
// --- PREPARATION ---
// Switch to (or create) a new database
use schoolDB

// Check which DB you are in
db

// --- 1. CREATE (using insertOne) ---
// (The question asks for insert(), but insertOne() is the modern standard)
db.students.insertOne({
  _id: 1,
  name: "Rajesh",
  major: "Computer Science",
  gpa: 8.5
})

// Insert another student
db.students.insertOne({
  _id: 2,
  name: "Priya",
  major: "Biology",
  gpa: 9.2
})

// --- 2. READ (using find) ---
```

```
// Find ALL documents in the 'students' collection
db.students.find()

// Find a specific document (Priya)
db.students.find({ name: "Priya" })

// Find all students with GPA > 9
db.students.find({ gpa: { $gt: 9 } })


// --- 3. UPDATE (using updateOne) ---
// (The question asks for update(), but updateOne() is the modern standard)
// We want to update Rajesh's GPA
// First argument: the query (WHO to update)
// Second argument: the change (WHAT to update)
db.students.updateOne(
  { name: "Rajesh" },
  { $set: { gpa: 8.7, major: "CSE" } }
)

// Check the result
db.students.find({ name: "Rajesh" })


// --- 4. DELETE (using deleteOne) ---
// (The question asks for remove(), but deleteOne() is the modern standard)
// Let's remove Rajesh
db.students.deleteOne({ name: "Rajesh" })

// Check the result
db.students.find()
```

🚀 Execution Steps

1. Make sure your MongoDB server is running.
2. Open your computer's terminal (like CMD or PowerShell).
3. Type `mongosh` and press Enter to start the MongoDB shell.
4. Type (or paste) the commands above one by one and press Enter to see the result of each.

📊 Sample Output

After `db.students.insertOne(...)`:

```
JSON
{ "acknowledged": true, "insertedId": 1 }
```

After `db.students.find()` (initial):

JSON
[
  { "_id": 1, "name": "Rajesh", "major": "Computer Science", "gpa": 8.5 },
  { "_id": 2, "name": "Priya", "major": "Biology", "gpa": 9.2 }
]

After `db.students.updateOne(...)`:

JSON
{ "acknowledged": true, "matchedCount": 1, "modifiedCount": 1 }

After `db.students.find({ name: "Rajesh" })`:

JSON
[
  { "_id": 1, "name": "Rajesh", "major": "CSE", "gpa": 8.7 }
]

After `db.students.deleteOne(...)`:

JSON
{ "acknowledged": true, "deletedCount": 1 }

After final `db.students.find()`:

JSON
[
  { "_id": 2, "name": "Priya", "major": "Biology", "gpa": 9.2 }
]

💡 Memory Tips

- **C**reate = `insertOne({ ... })`
- **R**ead = `find({ ... })`
- **U**pdate = `updateOne({ query }, { $set: { changes } })`
- **D**elete = `deleteOne({ ... })`

---

6. (a) ExpressJS: HTTP Methods (GET, POST, DELETE)

🎯 Aim

To write an Express program that uses different HTTP methods (verbs) to manipulate a resource:

- `GET` (retrieve data)
- `POST` (accept new data)
- `DELETE` (delete a specified resource)

📚 Description

RESTful APIs use HTTP methods to define actions. This program simulates a "To-Do" list API. We will use a simple array as our "database."

- `GET /todos`: Will retrieve the list of all to-do items.
- `POST /todos`: Will accept new data (a new to-do) and add it to the list.
- `DELETE /todos/:id`: Will find a to-do by its ID and remove it from the list.

We use the `express.json()` middleware to parse incoming JSON data on `POST` requests.

💻 Program (`index.js`)

JavaScript

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware to parse incoming JSON data
app.use(express.json());

// Our "database"
let todos = [
  { id: 1, task: "Buy milk" },
  { id: 2, task: "Code Express app" },
  { id: 3, task: "Walk the dog" }
];
let nextId = 4;

// 1. (GET) Retrieve all data
app.get('/todos', (req, res) => {
  res.json(todos);
});

// 2. (POST) Accept new data
app.post('/todos', (req, res) => {
  // Data comes in req.body
  const newTask = req.body.task;
  if (!newTask) {
```

```
    return res.status(400).json({ error: 'Task is required' });
  }

  const newTodo = {
    id: nextId++,
    task: newTask
  };

  todos.push(newTodo);
  res.status(201).json(newTodo); // 201 = Created
});

// 3. (DELETE) Delete a specified resource
app.delete('/todos/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const todoIndex = todos.findIndex(t => t.id === id);

  if (todoIndex === -1) {
    // Not found
    return res.status(404).json({ error: 'Todo not found' });
  }

  // Remove the item from the array
  todos.splice(todoIndex, 1);

  res.status(204).send(); // 204 = No Content (success)
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

🚀 Execution Steps

1. Create a folder (e.g., `lab-6a`).
2. `npm init -y`.
3. Install Express: `npm install express`.
4. Create `index.js` and paste the code.
5. Run the server: `node index.js`.
6. **Use Postman** (or a similar API tool) to test the routes. You cannot test `POST` or `DELETE` from a browser bar.

📊 Sample Output (Using Postman)

- **`GET` http://localhost:3000/todos**

Response:
JSON
```
[
  { "id": 1, "task": "Buy milk" },
  { "id": 2, "task": "Code Express app" },
  { "id": 3, "task": "Walk the dog" }
]
```

    -

- **`POST` http://localhost:3000/todos**
    - **Body (JSON):** `{ "task": "Water plants" }`

Response:
JSON
```
{ "id": 4, "task": "Water plants" }
```

    -

- **`DELETE` http://localhost:3000/todos/2**
    - **Response:** `Status 204 No Content`
- **`GET` http://localhost:3000/todos** (running again)

Response:
JSON
```
[
  { "id": 1, "task": "Buy milk" },
  { "id": 3, "task": "Walk the dog" },
  { "id": 4, "task": "Water plants" }
]
```

    -

💡 Memory Tips

- `GET` = `app.get(...)` (Read)
- `POST` = `app.post(...)` (Create, data in `req.body`)
- `DELETE` = `app.delete(...)` (Delete, ID in `req.params`)
- Use `express.json()` to read `req.body` from `POST` requests.

---

6. (b) MongoDB: Create and Drop Databases and Collections

🎯 Aim

To write `mongosh` queries to create and drop databases and collections.

📚 Description

In MongoDB:

- A **Database** is a container for collections.
- A **Collection** is a container for documents (functionally similar to a "table" in SQL).

MongoDB creates databases and collections "lazily." This means:

1. Running `use myNewDB` *selects* that database, but it isn't *actually* created on disk until you insert data.
2. You can, however, *explicitly* create a collection using `db.createCollection()`.

To drop a database, you **must** be `use`ing it first.

💻 Program (Queries)

JavaScript

```
// --- 1. Create/Use a Database ---
// This command switches your shell's context to 'companyDB'.
// If it doesn't exist, it will be created when you first add data.
use companyDB

// --- 2. Create a Collection Explicitly ---
// This will create 'companyDB' on disk (if it's not already)
// and create an empty 'employees' collection inside it.
db.createCollection("employees")

// You can verify it was created
show collections

// --- 3. Drop a Collection ---
// This removes the 'employees' collection and all its data.
db.employees.drop()

// Verify it's gone
show collections

// --- 4. Drop a Database ---
// To drop a database, you MUST be using it.
// We are already using 'companyDB' from step 1.
// WARNING: This deletes the entire database and all its collections!
db.dropDatabase()

// If you run 'show dbs', 'companyDB' will be gone.
show dbs
```

🚀 Execution Steps

1. Make sure your MongoDB server is running.
2. Open your terminal and type `mongosh`.
3. Enter the commands above one by one.

📊 Sample Output

After `use companyDB`:

switched to db companyDB

After `db.createCollection("employees")`:

{ "ok": 1 }

After `show collections`:

employees

After `db.employees.drop()`:

true

After `show collections`:

(empty output)

After `db.dropDatabase()`:

{ "dropped": "companyDB", "ok": 1 }

💡 Memory Tips

- `use [DB_NAME]` = Select database.
- `db.createCollection("[COLLECTION_NAME]")` = Create collection.
- `db.[COLLECTION_NAME].drop()` = Drop a single collection.
- `db.dropDatabase()` = Drop the *entire current* database. (Be careful!)

---

7. (a) ReactJS: Conditional Rendering

🎯 Aim

To write a React JS program that shows or hides UI elements based on a condition (e.g., if a user is logged in).

📚 Description

Conditional rendering is how we display different content based on state or props. The two most common ways to do this *inside* your JSX are:

1. **Ternary Operator (`? :`)**: This is the "if-else" for JSX. `condition ? <ShowThis /> : <ShowThat />`.
2. **Logical AND (`&&`)**: This is the "if-then" for JSX. `condition && <ShowThis />`. (It will render `<ShowThis />` *only if* the condition is true).

This program will have a "login" state, showing a "Welcome" message if logged in, and a "Please Log In" message if not.

💻 Program (`App.js`)

JavaScript

```
import React, { useState } from 'react';
import './App.css';

// A component to show when logged in
function WelcomeUser() {
  return (
    <div>
      <h2>Welcome Back, User!</h2>
      <p>Here is your dashboard.</p>
    </div>
  );
}

// A component to show when logged out
function WelcomeGuest() {
  return (
    <div>
      <h2>Please Log In</h2>
      <p>Sign in to access your dashboard.</p>
    </div>
  );
}

function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  const toggleLogin = () => {
    setIsLoggedIn(!isLoggedIn); // Toggle the boolean state
  };
```

```
  return (
    <div className="App">
      <h1>Conditional Rendering</h1>

      {/* 1. Using the Ternary Operator (if/else) */}
      {isLoggedIn ? <WelcomeUser /> : <WelcomeGuest />}

      {/* This button's text will also change based on the condition
      */}
      <button onClick={toggleLogin}>
        {isLoggedIn ? 'Log Out' : 'Log In'}
      </button>

      <hr />

      {/* 2. Using the Logical && Operator (if/then) */}
      {/* This "Admin Link" will ONLY be shown if isLoggedIn is true.
        If it's false, nothing is rendered.
      */}
      {isLoggedIn && (
        <a href="/admin">Go to Admin Panel</a>
      )}

    </div>
  );
}

export default App;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. `cd my-app`.
3. Replace `src/App.js` with the code above.
4. Start the app: `npm start`.

📊 Sample Output

Initially (isLoggedIn = false):

> **Conditional Rendering**
>
> **Please Log In** Sign in to access your dashboard.
>
> [Log In]

After clicking the "Log In" button (isLoggedIn = true):

> **Conditional Rendering**
>
> **Welcome Back, User!** Here is your dashboard.
>
> [Log Out]
>
> ---
>
> Go to Admin Panel

💡 Memory Tips

- **If/Else:** Use the **Ternary Operator** `? :`
  - `{condition ? <IfTrue /> : <IfFalse />}`
- **If-Then:** Use the **Logical AND** `&&`
  - `{condition && <IfTrue />}`

---

7. (b) ReactJS: Rendering Lists

🎯 Aim

To write a React JS program that renders a dynamic list of items from an array.

📚 Description

We *never* write lists manually in React if the data is dynamic. Instead, we use the JavaScript `.map()` array method.

1. Start with an array of data (e.g., an array of objects).
2. Inside your JSX, use `{}` to enter JavaScript mode.
3. Call `.map()` on your array.
4. For each item in the array, the `.map()` function should *return* a JSX element (like an `<li>`).
5. **CRITICAL:** You must provide a unique `key` prop to the *outermost* element you return (e.g., `<li key={item.id}>`). This helps React identify which item is which, which is crucial for performance and avoiding bugs.

💻 Program (`App.js`)

JavaScript
```
import React from 'react';
import './App.css';

// Our array of data
```

```
const fruits = [
  { id: 'f1', name: 'Apple', color: 'Red' },
  { id: 'f2', name: 'Banana', color: 'Yellow' },
  { id: 'f3', name: 'Kiwi', color: 'Green' },
  { id: 'f4', name: 'Blueberry', color: 'Blue' }
];

function App() {

  // Use .map() to transform the data array into a JSX array
  const fruitListItems = fruits.map(fruit => (
    // The 'key' prop MUST be here.
    // It should be a unique string or number (like an ID).
    <li key={fruit.id} style={{ color: fruit.color }}>
      {fruit.name}
    </li>
  ));

  return (
    <div className="App">
      <h1>List of Fruits</h1>

      <ul>
        {/* Render the array of JSX elements here */}
        {fruitListItems}
      </ul>

      {/* You can also do the .map() inline:
      */}
      <h2>Fruit Colors (Inline Map)</h2>
      <ul>
        {fruits.map(fruit => (
          <li key={fruit.id}>
            The color of {fruit.name} is {fruit.color}.
          </li>
        ))}
      </ul>

    </div>
  );
}

export default App;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.

2. `cd my-app`.
3. Replace `src/App.js` with the code above.
4. Start the app: `npm start`.

📊 Sample Output

The browser will show two lists:

List of Fruits

- Apple (in red text)
- Banana (in yellow text)
- Kiwi (in green text)
- Blueberry (in blue text)

Fruit Colors (Inline Map)

- The color of Apple is Red.
- The color of Banana is Yellow.
- The color of Kiwi is Green.
- The color of Blueberry is Blue.

💡 Memory Tips

- **Array -> JSX:** Use `.map()`.
- `{array.map(item => <li key={item.id}>{item.name}</li>)}`
- **`key` is King:** Always add a unique `key` prop to the top-level element inside your `.map()`.

---

8. (a) ExpressJS: Work with Form Data

🎯 Aim

To write an ExpressJS program that receives data from a standard HTML form (`application/x-www-form-urlencoded`).

📚 Description

When an HTML form is submitted, the browser packages the data and `POST`s it to the server. To make Express *understand* this data format, we must use a middleware: `express.urlencoded({ extended: true })`.

This middleware parses the incoming form data and attaches it as a JavaScript object to `req.body`.

This program will:

1. Serve a basic HTML form on the `GET /` route.
2. Listen for a `POST` request on the `/submit` route.
3. Use the middleware to read the data from `req.body` and send a response.

💻 Program (`index.js`)

JavaScript

```
const express = require('express');
const app = express();
const port = 3000;

// 1. Use the middleware for URL-encoded form data
app.use(express.urlencoded({ extended: true }));

// 2. Serve the HTML form
app.get('/', (req, res) => {
  res.send(`
    <h1>Simple HTML Form</h1>
    <form action="/submit" method="POST">
      <div>
        <label for="name">Name:</label>
        <input id="name" name="username">
      </div>
      <div>
        <label for="email">Email:</label>
        <input id="email" name="user_email">
      </div>
      <button type="submit">Submit Form</button>
    </form>
  `);
});

// 3. Handle the form submission
app.post('/submit', (req, res) => {
  // 4. The middleware puts the form data in req.body
  const name = req.body.username;
  const email = req.body.user_email;

  res.send(`
    <h1>Submission Received!</h1>
    <p>Thank you, ${name}.</p>
    <p>We received your email: ${email}</p>
  `);
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

🚀 Execution Steps

1. Create a folder (e.g., `lab-8a`).
2. `npm init -y`.
3. Install Express: `npm install express`.
4. Create `index.js` and paste the code.
5. Run the server: `node index.js`.
6. Visit `http://localhost:3000` in your browser.

📊 Sample Output

1. **Browser (visiting `/`)**: Shows the HTML form.
2. **You fill it out:** Name: "Student", Email: "student@test.com"
3. **You click "Submit Form"**: The browser makes a `POST` request.
4. **Browser (on `/submit`)**: Shows the response from the server:
   Submission Received! Thank you, Student. We received your email: student@test.com

💡 Memory Tips

- **Form Data:** `application/x-www-form-urlencoded`
- **Middleware:** `app.use(express.urlencoded({ extended: true }))`
- **Access Data:** `req.body`
- (For JSON data, you use `express.json()`).

---

8. (b) ReactJS: Writing Markup with JSX

🎯 Aim

To write a React JS program that demonstrates the key features and syntax of JSX.

📚 Description

JSX (JavaScript XML) looks like HTML, but it's actually a JavaScript syntax extension. It allows us to write our UI components declaratively. A "transpiler" (like Babel) converts this JSX into regular `React.createElement()` JavaScript calls.

Key differences from HTML:

1. **`class` -> `className`**: Because `class` is a reserved word in JavaScript.
2. **`for` -> `htmlFor`**: Because `for` is also a reserved word.

3. **JavaScript in `{}`**: You can embed any JavaScript expression (variables, function calls) inside curly braces.
4. **Inline Styles**: Styles are JavaScript objects, not strings. `style={{ color: 'blue' }}`.
5. **Single Root Element**: A component must return a *single* element. If you need more, wrap them in a **Fragment** (`<>...</>`) or a `div`.

💻 Program (`App.js`)

JavaScript

```
import React from 'react';
import './App.css';

function App() {
  // 1. JavaScript variable to embed
  const userName = "Student";
  const userRole = "Developer";

  // 2. JavaScript object for inline styles
  const headerStyle = {
    color: 'navy',
    fontSize: '32px',
    borderBottom: '2px solid navy'
  };

  // 3. A function to call
  const getGreeting = () => {
    return `Welcome, ${userName} (${userRole})!`;
  };

  return (
    // 4. Using a Fragment <> ... </> as the single root element
    <>
      {/* 2. Applying inline styles */}
      <h1 style={headerStyle}>JSX Demonstration</h1>

      {/* 1. & 3. Embedding JS expressions and function calls */}
      <p>{getGreeting()}</p>
      <p>Your lucky number is: {Math.floor(Math.random() * 100)}</p>

      {/* 5. Using className and htmlFor */}
      <div className="form-group">
        <label htmlFor="name-input">Name:</label>
        <input id="name-input" type="text" placeholder="This uses 'className'" />
      </div>
    </>
  );
```

}

export default App;

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. `cd my-app`.
3. Replace `src/App.js` with the code above.
4. Start the app: `npm start`.

📊 Sample Output

The browser will show:

> **JSX Demonstration** (as a large, navy, underlined heading)
>
> Welcome, Student (Developer)!
>
> Your lucky number is: [a random number]
>
> Name: [an input box]

💡 Memory Tips

- HTML `class` = JSX `className`
- HTML `for` = JSX `htmlFor`
- HTML `style="color:red;"` = JSX `style={{ color: 'red' }}`
- To use JS, use `{ }`.
- Always return a *single* root element (or use a Fragment `<>...</>`).

---

9. (a) ReactJS: Routing (React Router)

🎯 Aim

To write a React JS program that uses `react-router-dom` to navigate between different "pages" (components) in a Single Page Application (SPA).

📚 Description

In an SPA, the page *never* actually reloads. Instead, `react-router-dom` intercepts URL changes and dynamically renders different components.

- `BrowserRouter`: Wraps the *entire app* (usually in `index.js`). It enables routing.

- `Routes`: A container that holds all the possible `Route` definitions.
- `Route`: Defines a single "page." It maps a `path` (like `/about`) to an `element` (like `<About />`).
- `Link`: The component used to create navigation links. It's better than `<a>` because it prevents a page reload.

💻 Program

`index.js` (Root file)

JavaScript
```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
// 1. Import BrowserRouter
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    {/* 2. Wrap the entire App in BrowserRouter */}
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

`App.js` (Main router setup)

JavaScript
```
import React from 'react';
import { Routes, Route, Link } from 'react-router-dom';
import Home from './Home';
import About from './About';
import './App.css';

function App() {
  return (
    <div>
      {/* 3. Navigation using <Link> */}
      <nav>
        <ul>
          <li><Link to="/">Home</Link></li>
          <li><Link to="/about">About</Link></li>
        </ul>
      </nav>
```

```
      <hr />

      {/* 4. Define the routes */}
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </div>
  );
}

export default App;
```

`Home.js` (A "page" component)

```
JavaScript
import React from 'react';
function Home() {
  return <h1>Home Page</h1>;
}
export default Home;
```

`About.js` (Another "page" component)

```
JavaScript
import React from 'react';
function About() {
  return <h1>About Page</h1>;
}
export default About;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. `cd my-app`.
3. **Install React Router:** `npm install react-router-dom`.
4. Replace `src/index.js` with the code above.
5. Replace `src/App.js` with the code above.
6. Create `src/Home.js` and `src/About.js` and paste their code.
7. Start the app: `npm start`.

📊 Sample Output

The browser shows the "Home Page" and the navigation links.

> Home | About
>
> ---
>
> **Home Page**

When you click the "About" link:

- The URL in the browser bar changes to `http://localhost:3000/about`.
- The content below the `nav` changes to "About Page."
- The page **does not reload**.

> Home | About
>
> ---
>
> **About Page**

💡 Memory Tips

- `npm install react-router-dom`.
- Wrap app in `<BrowserRouter>` (in `index.js`).
- `<a>` becomes `<Link to="/path">`.
- `<Routes>` contains all `<Route>`.
- `<Route path="/path" element={<Component />} />` defines the page.

---

9. (b) ReactJS: Updating the Screen

🎯 Aim

To write a React JS program that updates what the user sees in response to an event.

📚 Description

This is the fundamental concept of React. You **never** update the screen (the DOM) directly. Instead:

1. You store the data you want to display in **state** (using `useState`).
2. You display that state in your JSX (e.g., `<h1>{myState}</h1>`).
3. When you want to update the screen, you call the **state setter function** (e.g., `setMyState("new value")`).
4. React "reacts" to this state change by *automatically* and *efficiently* re-rendering your component, updating the `<h1>` with the new value.

💻 Program (`App.js`)

JavaScript

```
import React, { useState } from 'react';
import './App.css';

function App() {
  // 1. Store the text in 'state'
  // 'text' holds the current value
  // 'setText' is the function to change it
  const [text, setText] = useState("This is the initial text.");

  // 2. Event handler that will update the state
  const handleUpdateClick = () => {
    // 3. Call the setter function to update the state
    setText("The screen has been updated!");
  };

  const handleResetClick = () => {
    setText("This is the initial text.");
  }

  return (
    <div className="App">
      <h1>Updating the Screen</h1>

      {/* This h2's content is tied directly to the 'text' state */}
      <h2>{text}</h2>

      <button onClick={handleUpdateClick}>
        Update the Text
      </button>
      <button onClick={handleResetClick}>
        Reset
      </button>
    </div>
  );
}

export default App;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. `cd my-app`.
3. Replace `src/App.js` with the code above.

4. Start the app: `npm start`.

📊 Sample Output

Initially:

> **Updating the Screen**
>
> This is the initial text.
>
> [Update the Text] [Reset]

After clicking "Update the Text":

> **Updating the Screen**
>
> The screen has been updated!
>
> [Update the Text] [Reset]

After clicking "Reset":

> **Updating the Screen**
>
> This is the initial text.
>
> [Update the Text] [Reset]

💡 Memory Tips

- To change what you see, change the **state**.
- `const [value, setValue] = useState(initialValue);`
- Call `setValue(newValue)` to trigger a re-render.
- **Never** do this: `value = "new value";` (This is wrong and won't work!).

---

10. (a) ReactJS: Creating and Nesting Components

🎯 Aim

To write a React JS program that defines both **Function** and **Class** components and shows how to **nest** them.

📚 Description

Components are the building blocks of React. They are reusable pieces of UI.

- **Function Components:** The modern standard. They are simple JavaScript functions that accept `props` and return JSX. They use **Hooks** (like `useState`) to manage state.
- **Class Components:** The older syntax. They are ES6 classes that `extend React.Component`. They use a `render()` method to return JSX and manage state using `this.state` and `this.setState()`.
- **Nesting:** Using one component inside another, just like you nest HTML tags (e.g., `<App>` uses `<Header>` and `<Profile>`).

💻 Program

`src/Header.js` (Function Component)

JavaScript
```
import React from 'react';

// This is a simple Function Component.
// It's just a function that returns JSX.
function Header() {
  return (
    <header style={{ background: '#eee', padding: '10px', textAlign: 'center' }}>
      <h1>My Application</h1>
    </header>
  );
}

export default Header;
```

`src/Profile.js` (Class Component)

JavaScript
```
import React from 'react';

// This is a Class Component.
// It needs to extend React.Component and have a render() method.
class Profile extends React.Component {
  render() {
    // It receives props via 'this.props'
    const name = this.props.userName;

    return (
      <div style={{ padding: '20px' }}>
        <h2>Welcome, {name}!</h2>
        <p>This component is a Class Component.</p>
      </div>
    );
```

```
  }
}

export default Profile;
```

`src/App.js` (Nesting Component)

```
JavaScript
import React from 'react';
import Header from './Header';     // Import the function component
import Profile from './Profile';  // Import the class component
import './App.css';

// App is a function component that NESTS the other two.
function App() {
  return (
    <div>
      {/* 1. Nesting the <Header> component */}
      <Header />

      {/* 2. Nesting the <Profile> component and passing a prop */}
      <Profile userName="Student" />

      <main style={{ padding: '20px' }}>
        <p>This is the main content of the App component.</p>
      </main>
    </div>
  );
}

export default App;
```

🚀 Execution Steps

1. Create a new React app: `npx create-react-app my-app`.
2. `cd my-app`.
3. Replace `src/App.js` with the code above.
4. Create `src/Header.js` and paste its code.
5. Create `src/Profile.js` and paste its code.
6. Start the app: `npm start`.

📊 Sample Output

The browser will show the content from all three components seamlessly combined:

My Application (from Header.js)

Welcome, Student! (from Profile.js) This component is a Class Component.

This is the main content of the App component. (from App.js)

💡 Memory Tips

- **Function:** `function MyComponent(props) { return ... }` (Modern, use Hooks).
- **Class:** `class MyComponent extends React.Component { render() { return ... } }` (Older, `this.props`, `this.state`).
- **Nesting:** Just use the component's name like an HTML tag: `<MyComponent />`.

---

10. (b) MongoDB: Advanced Queries

🎯 Aim

To write `mongosh` queries using `find()`, `limit()`, `sort()`, `createIndex()`, and `aggregate()`.

📚 Description

These commands allow for more complex data retrieval and optimization.

- `find()`: We know this; it's the base query.
- `.limit(N)`: Chained after `find()`, it restricts the result to `N` documents.
- `.sort({ field: 1 })`: Chained after `find()`, it sorts the results. `1` is for ascending, `-1` is for descending.
- `createIndex({ field: 1 })`: Creates an index on a field. This makes *searching* (using `find`) on that field much, much faster.
- `aggregate([...])`: The most powerful. It runs data through a "pipeline" of stages (like `$match`, `$group`, `$sort`) to perform complex transformations, like grouping data to calculate an average.

💻 Program (Queries)

JavaScript

```
// --- PREPARATION ---
use storeDB

// Insert sample data
db.products.insertMany([
  { name: "Laptop", category: "Electronics", price: 1200 },
  { name: "Mouse", category: "Electronics", price: 40 },
```

```
  { name: "Shirt", category: "Apparel", price: 50 },
  { name: "T-Shirt", category: "Apparel", price: 20 },
  { name: "Headphones", category: "Electronics", price: 150 }
])

// --- 1. find() ---
// Find all products
db.products.find()

// --- 2. limit() ---
// Find all products, but only return the first 2
db.products.find().limit(2)

// --- 3. sort() ---
// Find all products, sort by price descending (most expensive first)
db.products.find().sort({ price: -1 })

// Find all products, sort by category ascending
db.products.find().sort({ category: 1 })


// --- 4. createIndex() ---
// Create an index on the 'category' field to speed up category searches
db.products.createIndex({ category: 1 })

// Now, finds like this will be much faster:
db.products.find({ category: "Apparel" })


// --- 5. aggregate() ---
// This is a complex query.
// GOAL: Find the average price of products in EACH category.
db.products.aggregate([
  // Stage 1: Group documents by their 'category'
  {
    $group: {
      _id: "$category", // The field to group by
      avgPrice: { $avg: "$price" }, // Calculate the average of the 'price' field
      itemCount: { $sum: 1 } // Count how many items are in each group
    }
  },
  // Stage 2: Sort the resulting groups by their average price
  {
    $sort: { avgPrice: -1 }
  }
])
```

🚀 Execution Steps

1. Make sure your MongoDB server is running.
2. Open your terminal and type `mongosh`.
3. Enter the commands above one by one.

📊 Sample Output

After `db.products.find().sort({ price: -1 })`:

JSON
```
[
  { "_id": ..., "name": "Laptop", "category": "Electronics", "price": 1200 },
  { "_id": ..., "name": "Headphones", "category": "Electronics", "price": 150 },
  { "_id": ..., "name": "Shirt", "category": "Apparel", "price": 50 },
  { "_id": ..., "name": "Mouse", "category": "Electronics", "price": 40 },
  { "_id": ..., "name": "T-Shirt", "category": "Apparel", "price": 20 }
]
```

After `db.products.createIndex(...)`:

JSON
```
{
  "numIndexesBefore": 1,
  "numIndexesAfter": 2,
  "createdCollectionAutomatically": false,
  "ok": 1
}
```

After the `aggregate()` query:

JSON
```
[
  { "_id": "Electronics", "avgPrice": 463.33, "itemCount": 3 },
  { "_id": "Apparel", "avgPrice": 35, "itemCount": 2 }
]
```

💡 Memory Tips

- `find()` is the base.
- `.limit()` and `.sort()` are *chained* after `find()`.
- `sort({ field: 1 })` = Ascending (A-Z, 1-10)
- `sort({ field: -1 })` = Descending (Z-A, 10-1)
- `createIndex()` makes `find()` fast.

- `aggregate()` is for advanced "pipelines" (e.g., grouping, averages).