LAB 3: Dependency Injection, API and basic ReactJS

I. Loose coupling and Dependency Injection

Dependency Injection (DI) is a design pattern used to achieve Inversion of Control (IoC) between classes or modules. Instead of creating dependencies directly within a module, DI allows dependencies to be passed in, improving flexibility and making testing easier.

Scenario: *UserService* needs a database connection but shouldn't know which type of database it's connecting to.

Without DI:

MySQLDatabaseService:

- This class is specific to MySQL. It has two methods: *connectMySQL* to establish a connection and *getUserDataMySQL* to fetch user data from a MySQL database.
- This design hard-codes both the connection and data-fetching logic specifically for MySQL.

```
class MySQLDatabaseService {
    connectMySQL() {
      return "Connected to MySQL Database";
    }

    getUserDataMySQL() {
      return "Fetched user data from MySQL Database";
    }
  }
```

SQLServerDatabaseService:

- Similar to *MySQLDatabaseService*, this class provides methods (*connectSQLServer* and *getUserDataSQLServer*) for connecting to an SQL Server database and fetching user data.
- Like the MySQL version, it's specific to SQL Server, meaning it won't work with other database types.

```
class SQLServerDatabaseService {
  connectSQLServer() {
    return "Connected to SQL Server Database";
  }

  getUserDataSQLServer() {
    return "Fetched user data from SQL Server Database";
  }
}
```

UserService:

- This class directly instantiates MySQLDatabaseService, creating a **tight dependency** on MySQL.
- Since it instantiates *MySQLDatabaseService* directly, switching to *SQLServerDatabaseService* requires modifying *UserService*.

```
class UserService {
  constructor() {
    this.databaseService = new MySQLDatabaseService(); // Direct dependency
  }
}
```

In the examples provided (*userService1*, *userService2*, *userService3*), *UserService* is intended to use both MySQL and SQL Server services.

```
const userService1 = new UserService();
console.log(userService1.databaseService.connectMySQL());         // "Connected to MySQL Database"
console.log(userService1.databaseService.getUserDataMySQL());     // "Fetched user data from MySQL Database"

const userService2 = new UserService();
console.log(userService2.databaseService.SQLServerDatabaseService());  // "Connected to MySQL Database"
console.log(userService2.databaseService.getUserDataSQLServer());      // "Fetched user data from MySQL Database"

const userService3 = new UserService();
console.log(userService3.databaseService.connectMySQL());         // "Connected to MySQL Database"
console.log(userService3.databaseService.getUserDataMySQL());     // "Fetched user data from MySQL Database"
```

However, this code will cause errors if you try to use SQL Server methods without modifying *UserService* to support *SQLServerDatabaseService*. For instance:

- *userService2.databaseService.SQLServerDatabaseService()* is incorrect because *databaseService* is always a *MySQLDatabaseService* instance and does not have SQL Server methods.

```
console.log(userService2.databaseService.SQLServerDatabaseService());  // "Connected to SQL Server Database"
                                        ^

TypeError: userService2.databaseService.SQLServerDatabaseService is not a function
```

- This results in hard-to-trace bugs, as *UserService* was designed to support only MySQL.

Here, we'll use DI to inject different database services based on the required type, allowing *UserService* to work with various databases without modifying its internal logic.

Step 1: Define the Interface

First, define a common interface (*DatabaseService*) that each database type must implement:

```
class DatabaseService {
  connect() {
    throw new Error("Method 'connect()' must be implemented.");
  }

  getUserData() {
    throw new Error("Method 'getUserData()' must be implemented.");
  }
}
```

Step 2: Create Concrete Implementations for MySQL and SQL Server

Each database type will implement the *DatabaseService* interface.

MySQL Database Implementation

```
class MySQLDatabaseService extends DatabaseService {
  connect() {
    return "Connected to MySQL Database";
  }

  getUserData() {
    return "Fetched user data from MySQL Database";
  }
}
```

SQL Server Database Implementation

```
class SQLServerDatabaseService extends DatabaseService {
  connect() {
    return "Connected to SQL Server Database";
  }

  getUserData() {
    return "Fetched user data from SQL Server Database";
  }
}
```

Step 3: Inject Dependency into UserService

The *UserService* will accept any implementation of *DatabaseService*, making it database-agnostic.

```
class UserService {
  constructor(databaseService) { // Inject dependency
    this.databaseService = databaseService;
  }

  connectToDatabase() {
    return this.databaseService.connect();
  }

  fetchUserData() {
    return this.databaseService.getUserData();
  }
}
```

Step 4: Usage with Dependency Injection

Now, we can easily switch between *MySQLDatabaseService* and *SQLServerDatabaseService* when creating *UserService*.

```
const mysqlDatabaseService = new MySQLDatabaseService();
const userServiceMySQL = new UserService(mysqlDatabaseService);

console.log(userServiceMySQL.connectToDatabase()); // "Connected to MySQL Database"
console.log(userServiceMySQL.fetchUserData());     // "Fetched user data from MySQL Database"

const sqlServerDatabaseService = new SQLServerDatabaseService();
const userServiceSQL = new UserService(sqlServerDatabaseService);

console.log(userServiceSQL.connectToDatabase()); // "Connected to SQL Server Database"
console.log(userServiceSQL.fetchUserData());     // "Fetched user data from SQL Server Database"
```

```
Connected to MySQL Database
Fetched user data from MySQL Database
Connected to SQL Server Database
Fetched user data from SQL Server Database
```

Advantages of Dependency Injection

1. **Loose Coupling**: *UserService* doesn't depend on a specific database type; it depends on the interface. This allows easy switching between MySQL and SQL Server.
2. **Easy Testing**: During testing, you can mock *DatabaseService* or provide a fake implementation to isolate UserService's logic.
3. **Flexibility and Scalability**: Adding support for a new database type (e.g., PostgreSQL) would require implementing *DatabaseService* for PostgreSQL and passing it to *UserService*.

Exercise:

Objective: Refactor the code to use DI, allowing *UserManager* to interact seamlessly with **any database service**.

1. **MySQLDatabaseService**: Has *connectMySQL* and *getUserDataMySQL* methods

```
class MySQLDatabaseService {
    connectMySQL() {
      console.log("Connected to MySQL Database");
    }

    getUserDataMySQL(userId) {
      console.log(`Fetched data for user ID ${userId} from MySQL Database.`);
      return { id: userId, name: "Alice (MySQL)" };
    }
  }
```

2. **SQLServerDatabaseService**: Has *connectSQLServer*, *getUserDataSQLServer*, and an additional *closeSQLServerConnection* method.

```
class SQLServerDatabaseService {
  connectSQLServer() {
    console.log("Connected to SQL Server Database");
  }

  getUserDataSQLServer(userId) {
    console.log(`Fetched data for user ID ${userId} from SQL Server Database.`);
    return { id: userId, name: "Bob (SQL Server)" };
  }

  closeSQLServerConnection() {
    console.log("Closed SQL Server Database connection");
  }
}
```

3. **PostgreSQLDatabaseService**: Has *connectPostgreSQL*, *getUserDataPostgreSQL*, and *rollbackTransaction* methods.

```
class PostgreSQLDatabaseService {
  connectPostgreSQL() {
    console.log("Connected to PostgreSQL Database");
  }

  getUserDataPostgreSQL(userId) {
    console.log(`Fetched data for user ID ${userId} from PostgreSQL Database.`);
    return { id: userId, name: "Charlie (PostgreSQL)" };
  }

  rollbackTransaction() {
    console.log("Rolled back PostgreSQL transaction");
  }
}
```

4. **UserManager**: Manages users by interacting directly with one of these services.

```
class UserManager {
  constructor() {
    this.databaseService = new MySQLDatabaseService(); // Hardcoded dependency
  }

  getUser(userId) {
    // Assuming we're working with MySQL here
    this.databaseService.connectMySQL();
    const user = this.databaseService.getUserDataMySQL(userId);
    console.log(`User Details:`, user);
    return user;
  }
}

// Usage Example
const userManager = new UserManager();
userManager.getUser(1);
```

II. Application Programming Interface (API)

```
[
  - {
      userId: 1,
      id: 1,
      title: "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
      body: "quia et suscipit
      suscipit recusandae consequuntur expedita et cum
      reprehenderit molestiae ut ut quas totam
      nostrum rerum est autem sunt rem eveniet architecto"
    },
  - {
      userId: 1,
      id: 2,
      title: "qui est esse",
      body: "est rerum tempore vitae
      sequi sint nihil reprehenderit dolor beatae ea dolores neque
      fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis
      qui aperiam non debitis possimus qui neque nisi nulla"
    },
```


HTML:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Fetch API Example</title>
</head>
<body>
    <h1>Posts</h1>
    <div id="posts-container"></div>

    <script src="app.js"></script>
</body>
</html>
```

Javascript:

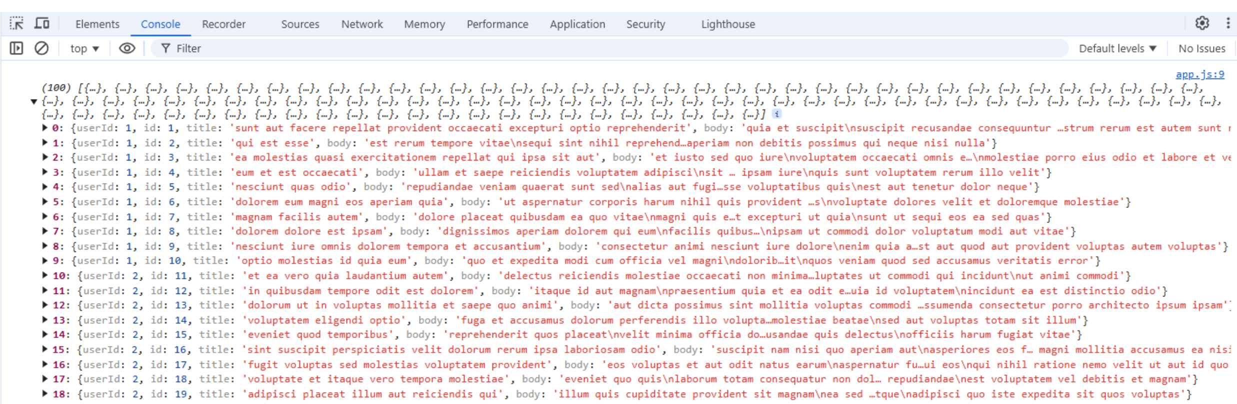
```javascript
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
    displayPosts(data);
  })
  .catch(error => console.error('There was a problem with the fetch operation:', error));

function displayPosts(posts) {
  const postsContainer = document.getElementById('posts-container');

  posts.forEach(post => {
    const postDiv = document.createElement('div');
    postDiv.classList.add('post');
    postDiv.innerHTML = `
      <h2>${post.title}</h2>
      <p>${post.body}</p>
    `;
    postsContainer.appendChild(postDiv);
  });
}
```

Explanation

1. **Fetching Data**:
    - We use *fetch()* to make a GET request to the API endpoint.
    - When the response is received, it’s converted to JSON.
    - The JSON data is passed to the displayPosts function.
2. **Displaying Data**:
    - The displayPosts function loops through each post, creating a new <div> element for each.
    - Each post is formatted with a title and body and appended to the posts-container in the HTML.
3. **Error Handling**:
    - We catch any network or data processing errors and log them to the console.

How to View the Console Log

1. **Open your web page**: Load your HTML file in a web browser.
2. **Open Developer Tools**:
    - Right-click on the page and select "Inspect" or press Ctrl + Shift + I (Windows/Linux) or Cmd + Option + I (Mac).
3. **Go to the Console Tab**: Click on the "Console" tab in the Developer Tools panel to see the logged data.



Result:

Parse to Object

To work with the data you fetched from the API, you'll typically parse the JSON into JavaScript objects. This allows you to manipulate the data easily, access specific properties, and implement methods for interaction.

Why Parse JSON?

1. **Conversion to Objects**: The raw response from the Fetch API is a string in JSON format. Parsing it converts this string into a JavaScript object (or array of objects), allowing you to easily access and manipulate the data.
2. **Data Manipulation**: Once the data is parsed, you can perform operations like filtering, sorting, or modifying properties.
3. **Dynamic Interactions**: By encapsulating the data in objects, you can create methods that provide functionalities, such as searching for posts, updating them, or deleting them.

```
class Post {
  constructor({ userId, id, title, body }) {
    this.userId = userId;
    this.id = id;
    this.title = title;
    this.body = body;
  }

  display() {
    return `
      <div class="post">
        <h2>${this.title}</h2>
        <p>${this.body}</p>
      </div>
    `;
  }

  update(data) {
    if (data.title) this.title = data.title;
    if (data.body) this.body = data.body;
  }
}
```

```
class PostManager {
  constructor(posts) {
    this.posts = posts; // Store posts
  }

  displayPosts() {
    const postsContainer = document.getElementById('posts-container');
    postsContainer.innerHTML = this.posts.map(post => post.display()).join('');
  }

  findPostById(id) {
    return this.posts.find(post => post.id === id);
  }

  filterPostsByUserId(userId) {
    return this.posts.filter(post => post.userId === userId);
  }

  updatePost(id, newData) {
    const post = this.findPostById(id);
    if (post) {
      post.update(newData);
      console.log(`Post ${id} updated.`);
    } else {
      console.log(`Post ${id} not found.`);
    }
  }
```

```
  deletePost(id) {
    const initialLength = this.posts.length;
    this.posts = this.posts.filter(post => post.id !== id);
    if (this.posts.length < initialLength) {
      console.log(`Post ${id} deleted.`);
    } else {
      console.log(`Post ${id} not found.`);
    }
  }
}
```

Additional Methods Explained

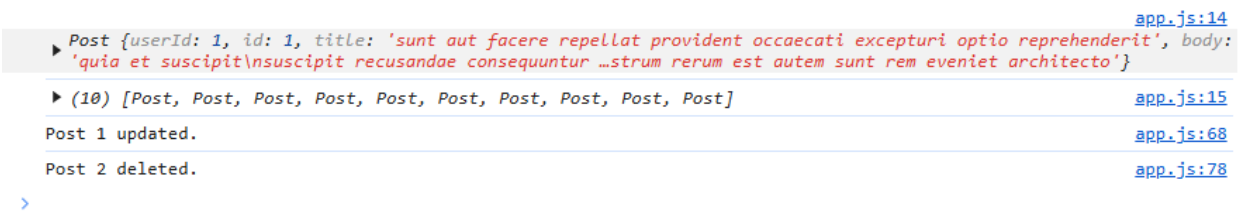
1. **Update Method in Post Class**:
    - **update(data)**: This method updates the properties of the Post instance. You can pass an object containing the properties you want to update (e.g., title or body).
2. **PostManager Class Methods**:
    - **filterPostsByUserId(userId)**: This method filters the posts by a specific user ID. It returns an array of posts that belong to the specified user.
    - **updatePost(id, newData)**: This method updates the post with the given ID. It uses the update method of the Post class to change the title and/or body of the post.
    - **deletePost(id)**: This method removes the post with the specified ID from the collection. It uses the filter method to create a new array without the deleted post. If a post is found and deleted, it logs a success message; otherwise, it logs that the post was not found.

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    const posts = data.map(postData => new Post(postData));
    const postManager = new PostManager(posts);
    postManager.displayPosts();

    console.log(postManager.findPostById(1));
    console.log(postManager.filterPostsByUserId(1));
    postManager.updatePost(1, { title: "Updated Title", body: "Updated body text." });
    postManager.deletePost(2);
    postManager.displayPosts();
  })
  .catch(error => console.error('There was a problem with the fetch operation:', error));
```

In the then block after fetching the data:

- **Finding a Post**: *console.log(postManager.findPostById(1));* will log the post with ID 1 to the console.

- **Filtering Posts**: *console.log(postManager.filterPostsByUserId(1));* will log all posts from user ID 1.
- **Updating a Post**: *postManager.updatePost(1, { title: "Updated Title", body: "Updated body text." });* will update the post with ID 1.
- **Deleting a Post**: *postManager.deletePost(2);* will delete the post with ID 2.



Exercise 1: Fetch and Analyze Post Data

Objective: Fetch post data from an API and perform basic analysis on it.

Steps:

1. **API Selection**: Use the *JSONPlaceholder* API to fetch post data.
2. **Create an HTML File**: Set up a simple HTML structure with a * * to display post statistics.
3. **Fetch Data**:
    - Use the Fetch API to get data from the */posts* endpoint.

- Parse the JSON response.

4. **Analyze Data**:
   - Calculate and display:
     - The total number of posts.
     - The average length of the post titles.
     - The number of posts created by a specific user (e.g., user ID 1).
5. **Display Analysis**:
   - Insert the calculated statistics into the HTML * *.

Expected Outcome: A web page that shows the total number of posts, average title length, and the number of posts created by user ID 1.

Exercise 2: **Fetch and Display User Information**

Objective: Fetch user data from a public API and display it on a web page.

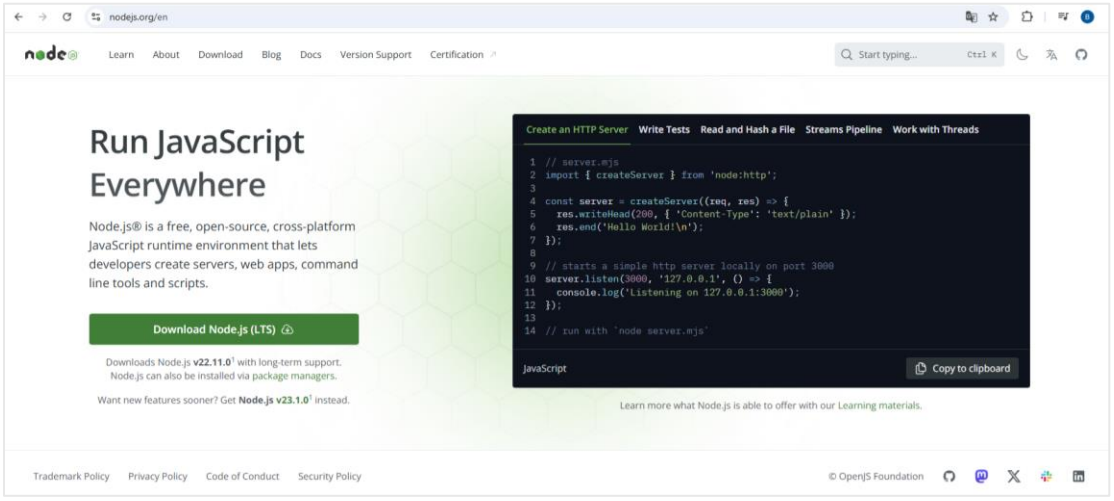
Steps:

1. **API Selection**: Use the JSONPlaceholder API to fetch user data.
2. **Create an HTML File**: Set up a basic HTML structure with a to display user information.
3. **Fetch Data**:
   - Use the Fetch API to get data from the /users endpoint.
   - Parse the JSON response.
4. **Display Data**:
   - Loop through the fetched user data and create a list of user names with their email addresses.
   - Insert the list into the HTML .
5. **Bonus**: Add a search feature that filters users based on their name.

Expected Outcome: A web page that displays a list of users along with their email addresses. Implementing the search feature will allow users to filter the displayed list by typing in a search box.

III. ReactJS

Before starting to code in ReactJS, you need to install **NodeJS** and **npm**. You can install them from the website *https://nodejs.org/en*.



Create new ReactJS project:

```
npx create-react-app <project_name>
```

```
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...
```

```
npm start
```

1. Components

Components are the building blocks of a React application. Each component is a JavaScript function or class that returns a piece of the user interface.

Code Example:

```jsx
function Greeting() {
  return <h1>Hello, World!</h1>;
}

function App() {
  return (
    <div>
      <Greeting />
    </div>
  );
}

export default App;
```

2. JSX

JSX stands for JavaScript XML. It allows you to write HTML-like syntax directly in JavaScript.

Code Example:

```jsx
function App() {
  const name = "React Learner";
  return <h1>Hello, {name}!</h1>;
}

export default App;
```

3. Props

Props (short for properties) are used to pass data from one component to another.

Code Example:

```jsx
function Profile(props) {
  return <h2>Hello, {props.name}!</h2>;
}

function App() {
  return <Profile name="Alice" />;
}

export default App;
```

4. State

State is used to manage dynamic data within a component.

Code Example:

```jsx
import { useState } from 'react';  4.2k (gzipped: 1.9k)

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

5. Event Handling

In React, events are handled using camelCase syntax and functions.

Code Example:

```
function ClickButton() {
  function handleClick() {
    alert('Button clicked!');
  }

  return <button onClick={handleClick}>Click Me</button>;
}
export default ClickButton;
```

6. Conditional Rendering

Conditional rendering in React allows you to render components or elements based on certain conditions.

Code Example:

```
function Message({ isLoggedIn }) {
  return isLoggedIn ? <h1>Welcome Back!</h1> : <h1>Please Sign Up</h1>;
}

export default Message;
```

7. Lists and Keys

Lists in React are used to render multiple items, and each item should have a unique key for efficient re-rendering.

Code Example:

```
function NameList() {
  const names = ['Alice', 'Bob', 'Charlie'];
  return (
    <ul>
      {names.map((name, index) => (
        <li key={index}>{name}</li>
      ))}
    </ul>
  );
}

export default NameList;
```

8. Hooks (useState, useEffect)

Hooks let you use state and other React features in functional components.

Code Example:

```
import { useState, useEffect } from 'react';  4.3k (gzipped: 1.9k)

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => setCount(c => c + 1), 1000);
    return () => clearInterval(timer);
  }, []);

  return <p>Timer: {count}</p>;
}
```

9. Context API

The Context API provides a way to pass data through the component tree without having to pass props manually at every level.

Code Example:

```
import React, { createContext, useContext } from 'react';  7k (gzipped: 2.8k)

const UserContext = createContext();

function Greeting() {
  const user = useContext(UserContext);
  return <h1>Hello, {user.name}!</h1>;
}

function App() {
  const user = { name: 'Alice' };

  return (
    <UserContext.Provider value={user}>
      <Greeting />
    </UserContext.Provider>
  );
}

export default App;
```

Example: To-Do List App with Context and Local State

Goal: Build a small to-do list application that allows users to:

1. Add new to-do items.
2. Toggle each to-do item as "complete" or "incomplete."
3. Filter to-do items by "all," "complete," or "incomplete."

The application will make use of:

- **Components** for modular code.
- **Props** to pass data between components.
- **State** to handle dynamic data like the to-do list.
- **Event Handling** for user interactions.
- **Conditional Rendering** to display filtered items.
- **Context API** to share the filter status across components.

Step 1: Set Up Context and Components

1. **Context API** - Use the Context API to create a shared filter state. This will allow all components to access and update the filter setting without needing to pass it through each component as a prop.
2. **Components and Props** - Organize your app into a few components: App, TodoList, TodoItem, and FilterControl. Use props to pass the necessary data and functions from App to the other components.

Step 2: Add To-Do Items

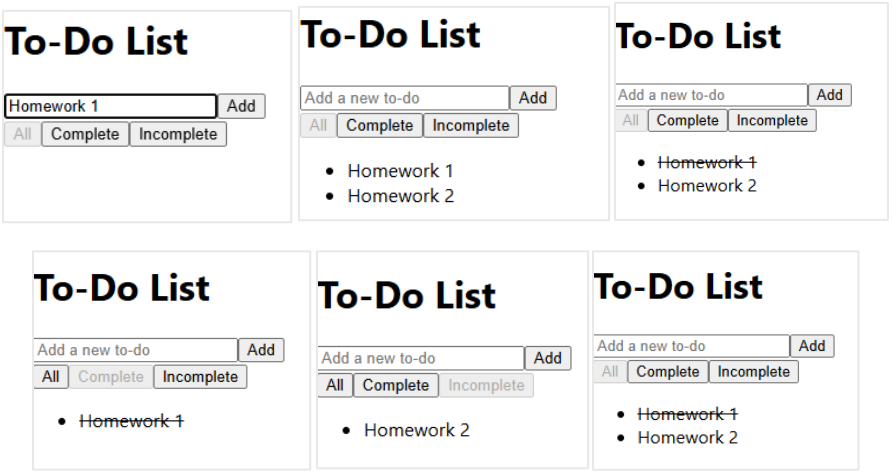
1. **State** - Use the useState hook to manage the list of to-dos within the App component. Each to-do should be an object with id, text, and completed properties.
2. **Event Handling** - Create an input field and a button for adding new to-do items. Set up an event handler that adds the to-do to the list and clears the input field.

Step 3: Filter and Toggle To-Do Items

1. **Conditional Rendering** - In TodoList, render the items based on the filter (show all, completed, or incomplete). Use Array.filter() to apply the filter.
2. **Context API and State** - Use Context to keep track of the current filter. In FilterControl, set up buttons to update the filter status (all, complete, incomplete).

Note on Concepts Used:

- **Context API**: Used here to avoid "prop drilling" (passing props through multiple components). Context allows us to share the filter setting directly with components that need it.
- **Components and Props**: Each part of the app is a component with a clear responsibility, making the app modular. Props pass data, such as each to-do item, from the parent to the child components.
- **State**: useState allows us to keep track of dynamic data like the list of to-dos and the filter setting.
- **Event Handling**: Event handlers are used to respond to user actions, like adding a new to-do or toggling a filter.
- **Conditional Rendering**: This enables displaying different data (e.g., filtered to-dos) based on conditions (the selected filter).



Create a *CartContext* to manage the cart state and add item to cart:

```
const CartContext = createContext();

function App() {
  const products = [
    { id: 1, name: 'Laptop', price: 999 },
    { id: 2, name: 'Smartphone', price: 699 },
    { id: 3, name: 'Headphones', price: 199 },
  ];

  const [cart, setCart] = useState([]);

  // Add item to cart
  const addToCart = (product) => {
    setCart((prevCart) => {
      const productInCart = prevCart.find(item => item.id === product.id);
      if (productInCart) {
        return prevCart.map(item =>
          item.id === product.id ? { ...item, quantity: item.quantity + 1 } : item
        );
      } else {
        return [...prevCart, { ...product, quantity: 1 }];
      }
    });
  };
```

Remove item from cart and return:

```
const removeFromCart = (productId) => {
  setCart((prevCart) => prevCart.filter(item => item.id !== productId));
};

return (
  <CartContext.Provider value={{ cart, addToCart, removeFromCart }}>
    <div className="App">
      <h1>Shopping Cart</h1>
      <ProductList products={products} />
      <Cart />
    </div>
  </CartContext.Provider>
);
```

Component for listing products:

```
function ProductList({ products }) {
  return (
    <div>
      <h2>Products</h2>
      {products.map(product => (
        <ProductItem key={product.id} product={product} />
      ))}
    </div>
  );
}
```

Component for each product item:

```
function ProductItem({ product }) {
  const { addToCart } = useContext(CartContext);

  return (
    <div style={{ marginBottom: '10px' }}>
      <p>{product.name} - ${product.price}</p>
      <button onClick={() => addToCart(product)}>Add to Cart</button>
    </div>
  );
}
```

Component to display items in the cart and calculate total price:

```
function Cart() {
  const { cart, removeFromCart } = useContext(CartContext);

  const total = cart.reduce((acc, item) => acc + item.price * item.quantity, 0);

  return (
    <div>
      <h2>Cart</h2>
      {cart.length === 0 ? (
        <p>The cart is empty</p>
      ) : (
        <>
          <ul>
            {cart.map(item => (
              <CartItem key={item.id} item={item} />
            ))}
          </ul>
          <h3>Total: ${total.toFixed(2)}</h3>
        </>
      )}
    </div>
  );
```

Component for each cart item:

```
function CartItem({ item }) {
  const { removeFromCart } = useContext(CartContext);

  return (
    <li>
      {item.name} - ${item.price} x {item.quantity}
      <button onClick={() => removeFromCart(item.id)}>Remove</button>
    </li>
  );
}

export default App;
```

Exercise: Shopping Cart App

Goal: Build a basic shopping cart application that allows users to:

1. View a list of products.
2. Add products to the cart.
3. Remove products from the cart.
4. View the total price of items in the cart.

This app will utilize:

- **Components** for modularity.
- **Props** to pass data between components.
- **State** to manage the product list and the cart.
- **Context API** to share the cart state across components.
- **Conditional Rendering** for the cart display.