INP-2025-May-PYQ Answers

Q1. [10 Marks]

a. Explain the difference between State and Props in ReactJS with an example.

b. What is REST API? Discuss its core principles and why they are important?

Q2. [10 Marks]

a. Explain Buffers and Streams in NodeJS. Provide an example where they are useful.

b. Discuss different phases of the React component lifecycle.

Q3. [10 Marks]

a. Discuss difference between var, let and const in ES6 with examples.

b. Explain promises with an example.

Q4. [10 Marks]

a. Explain File System module of NodeJS in detail.

b. What is web browser. Discuss working of web browser in detail.

Q5. [10 Marks]

a. What is Express JS. Explain features of Express JS.

b. Explain Flux architecture in detail.

Q6. [20 Marks]

a. Write a short note on any 4

- ES5 vs ES6
- JSX

- JSON
- URL vs URI
- Arrow Function

Q1. [10 Marks] - Answers

a. Explain the difference between State and Props in ReactJS with an example.

State vs Props in ReactJS

Introduction

In ReactJS, **State** and **Props** are two core concepts used to manage data within components.

- **State** handles **dynamic, internal data** that can change during the component's lifecycle.
- **Props** are **read-only attributes** used to pass data from a **parent** component to a **child** component.

Feature	**State**	**Props**
Definition	Internal data managed by the component itself	External data passed from parent to child component
Mutability	Mutable (can be changed using `setState` or hooks)	Immutable (read-only inside the child component)
Ownership	Owned and controlled by the component	Owned by parent, received by child
Usage Purpose	Used to handle dynamic data and user interactions	Used to pass data and configuration to child components
Modification	Modified within the component	Cannot be modified by the receiving component

Feature	State	Props
Lifecycle Impact	Changes in state trigger re-rendering	Changes in props also trigger re-rendering
Access Method	Accessed using `this.state` (class) or `useState` (hooks)	Accessed via `this.props` (class) or directly (hooks)
Example Use Case	Toggle button, form input, modal visibility	Displaying user name, passing theme or styles

Example:

```
// Parent Component
function Parent() {
  return <Child name="Sameer" />;
}

// Child Component using Props
function Child(props) {
  return <h1>Hello, {props.name}</h1>;
}

// Component using State
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

b. What is REST API? Discuss its core principles and why they are important?

What is REST API?

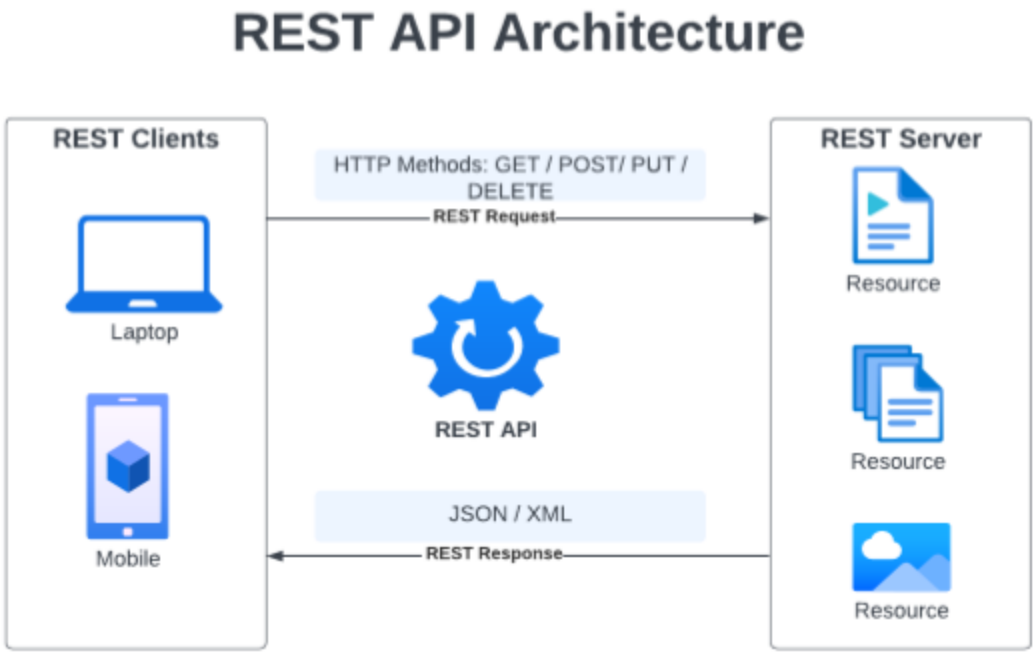
- **REST (Representational State Transfer)** is an architectural style for designing networked applications.
- A **REST API** allows communication between client and server using standard HTTP methods (GET, POST, PUT, DELETE).
- It treats server-side resources as accessible via **URIs**, and interactions are stateless.

Core Principles of REST

Principle	Description
Statelessness	Each request from client to server must contain all necessary information.
Client-Server	Separation of concerns: UI (client) and data storage (server) are independent.
Uniform Interface	Consistent way to interact with resources using standard HTTP methods.
Representation	Resources are represented in formats like JSON or XML.
Cacheability	Responses must define whether they can be cached to improve performance.
Layered System	Architecture can have multiple layers (e.g., proxy, gateway) for scalability.

Why These Principles Are Important

- **Scalability**: Statelessness and layered architecture allow systems to scale easily.
- **Maintainability**: Clear separation between client and server simplifies updates and debugging.
- **Interoperability**: Uniform interface and resource-based design make APIs easier to consume across platforms.
- **Performance**: Caching reduces server load and improves response time.



Q2. [10 Marks] - Answers

a. Explain Buffers and Streams in NodeJS. Provide an example where they are useful.

Buffers in Node.js

- **Definition**: Buffers are temporary memory allocations used to store binary data.
- **Use Case**: Useful when dealing with raw binary data like files, images, or network packets.
- **Characteristics**:
  - Fixed size
  - Used when data is not encoded as strings (e.g., TCP streams, file I/O)
  - Created using `Buffer.from()`, `Buffer.alloc()`, etc.

```
const buf = Buffer.from('Hello');
console.log(buf); // <Buffer 48 65 6c 6c 6f>
```

Streams in Node.js

- **Definition**: Streams are objects that let you read or write data **piece by piece** (chunks) instead of all at once.
- **Types**:
  - **Readable**: Read data (e.g., `fs.createReadStream`)
  - **Writable**: Write data (e.g., `fs.createWriteStream`)
  - **Duplex**: Both readable and writable (e.g., TCP sockets)
  - **Transform**: Modify data as it passes through (e.g., compression)
- **Benefits**:
  - Efficient memory usage
  - Ideal for large files or continuous data (e.g., video streaming)

```
const fs = require('fs');
const readStream = fs.createReadStream('input.txt');
readStream.on('data', chunk => {
  console.log('Received chunk:', chunk);
});
```

Example Use Case: File Upload Handling

- **Scenario**: A user uploads a large video file to a server.
- **Why Buffers & Streams?**
  - **Buffer**: Temporarily holds chunks of binary data.

    - **Stream**: Processes the file in chunks without loading the entire file into memory.
- **Result**: Faster uploads, reduced memory usage, and better scalability.

b. Discuss different phases of the React component lifecycle.

React Component Lifecycle Overview

React components go through three main phases during their existence:

1. **Mounting Phase**

This phase occurs when the component is created and inserted into the DOM.

- `constructor()` – Initializes state and binds methods (class components only).
- `render()` – Returns JSX to display the UI.
- `componentDidMount()` – Executes after the component is mounted; ideal for API calls or DOM setup.
- In functional components, `useEffect(() => { ... }, [])` serves the same purpose as `componentDidMount`.

2. **Updating Phase**

Triggered when props or state change, causing the component to re-render.

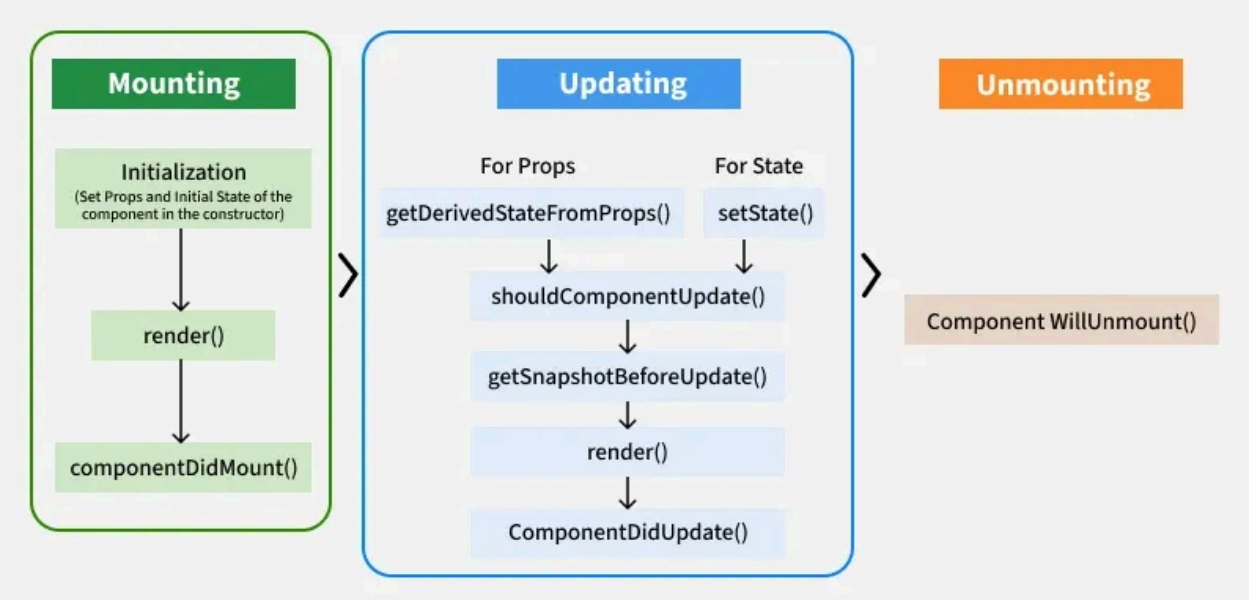
- `shouldComponentUpdate()` – Determines whether the component should re-render (used for optimization).
- `render()` – Re-renders the component with updated data.
- `componentDidUpdate(prevProps, prevState)` – Runs after the update; useful for reacting to changes.
- Functional components use `useEffect(() => { ... }, [dependencies])` to handle updates.

3. **Unmounting Phase**

Occurs when the component is removed from the DOM.

- `componentWillUnmount()` – Used for cleanup tasks like removing event listeners or timers.
- In functional components, cleanup is done using `useEffect(() => { return () => { ... } }, [])`.

React component lifecycle diagram:



Q3. [10 Marks] - Answers

a. Discuss difference between var, let and const in ES6 with examples.

Key Differences

1. **Scope**

- `var` is **function-scoped**: accessible throughout the function where it's declared.

- `let` and `const` are **block-scoped**: accessible only within the `{}` block where they are defined.

```
function test() {
  if (true) {
    var x = 10;
    let y = 20;
    const z = 30;
  }
  console.log(x); // 10
  console.log(y); // Error: y is not defined
  console.log(z); // Error: z is not defined
}
```

2. **Re-declaration**

- `var` allows re-declaration in the same scope.
- `let` and `const` do **not** allow re-declaration in the same scope.

```
var a = 5;
var a = 10; // Allowed

let b = 5;
let b = 10; // Error

const c = 5;
const c = 10; // Error
```

3. **Re-assignment**

- `var` and `let` can be re-assigned.
- `const` **cannot** be re-assigned after initialization.

```
let score = 100;
score = 200; // Allowed

const pi = 3.14;
pi = 3.1415; // Error: Assignment to constant variable
```

4. **Hoisting**

- All three are hoisted, but:
  - `var` is initialized as `undefined`.
  - `let` and `const` are in a **temporal dead zone** until declared.

```
console.log(a); // undefined
var a = 10;

console.log(b); // Error
let b = 20;
```

b. Explain promises with an example.

What is a Promise?

- A **Promise** is an object in JavaScript that represents the eventual **completion (or failure)** of an asynchronous operation and its resulting value.
- It helps manage asynchronous code in a cleaner and more readable way than nested callbacks (callback hell).

Promise States

1. **Pending** – Initial state, neither fulfilled nor rejected.
2. **Fulfilled** – Operation completed successfully.
3. **Rejected** – Operation failed with an error.

Syntax:

```
const promise = new Promise((resolve, reject) => {
  // async operation
  if (success) {
    resolve(result); // fulfilled
  } else {
    reject(error);   // rejected
  }
});
```

Example: Simulating Data Fetch

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Data loaded successfully");
      } else {
        reject("Failed to load data");
      }
    }, 1000);
  });
}

fetchData()
```

```
  .then(result ⇒ console.log(result))     // Output: Data loaded successfully
  .catch(error ⇒ console.error(error));   // Handles any error
```

Q4. [10 Marks] - Answers

a. Explain File System module of NodeJS in detail.

Node.js File System (fs) Module

- The `fs` **module** is a **core Node.js module** used to **interact with the file system**.
- You can **read, write, update, delete, and manage files or directories**.
- It provides both **synchronous** and **asynchronous** methods.

Key Methods

Method	Description
`fs.readFile()`	Read the contents of a file (async)
`fs.readFileSync()`	Read a file synchronously
`fs.writeFile()`	Create/write data to a file (async)
`fs.writeFileSync()`	Create/write file synchronously
`fs.appendFile()`	Add data to an existing file
`fs.unlink()`	Delete a file
`fs.mkdir()`	Create a new directory

Example:

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

// 1️⃣ Create a new file and add data
fs.writeFile('sample.txt', 'This is the first line.\n', (err) ⇒ {
  if (err) throw err;
```

```
  console.log('File created and data added.');

  // 2️⃣ Append more data
  fs.appendFile('sample.txt', 'This is the appended line.\n', (err) => {
    if (err) throw err;
    console.log('Data appended successfully.');

    // 3️⃣ Read file data (without buffer)
    fs.readFile('sample.txt', 'utf8', (err, data) => {
      if (err) throw err;
      console.log('File content:\n', data);

      // 4️⃣ Rename the file
      fs.rename('sample.txt', 'renamedSample.txt', (err) => {
        if (err) throw err;
        console.log('File renamed successfully.');

        // 5️⃣ Delete the file
        fs.unlink('renamedSample.txt', (err) => {
          if (err) throw err;
          console.log('File deleted successfully.');
        });
      });
    });
  });
});
```

b. What is web browser. Discuss working of web browser in detail.

What is a Web Browser?

- A **web browser** is a software application used to access, retrieve, and display content from the **World Wide Web**.

- It communicates with web servers using the **HTTP/HTTPS protocols** and renders HTML, CSS, JavaScript, and other web technologies.
- Examples include **Google Chrome**, **Mozilla Firefox**, **Microsoft Edge**, and **Safari**.

Working of a Web Browser (Step-by-Step)

1. **User Input (URL or Search Query)**

- The user enters a **URL** (e.g., `https://example.com`) or a **search term**.
- The browser checks if the input is a valid web address or needs to be searched via a search engine.

2. **DNS Resolution**

- The browser contacts a **DNS server** to convert the domain name (e.g., `example.com`) into an **IP address** of the web server.

3. **Establishing Connection**

- The browser initiates a **TCP connection** with the server.
- If the URL uses HTTPS, a **TLS/SSL handshake** is performed for secure communication.

4. **Sending HTTP Request**

- The browser sends an **HTTP GET request** to the server asking for the web page or resource.

5. **Receiving HTTP Response**

- The server responds with an **HTTP response**, which includes:
  - **Status code** (e.g., 200 OK, 404 Not Found)
  - **Headers** (metadata)

- **Body** (HTML content)

6. **Rendering the Page**

- The browser parses the **HTML** and builds the **DOM (Document Object Model)**.
- It loads **CSS** for styling and **JavaScript** for interactivity.
- External resources (images, fonts, scripts) are fetched and rendered.

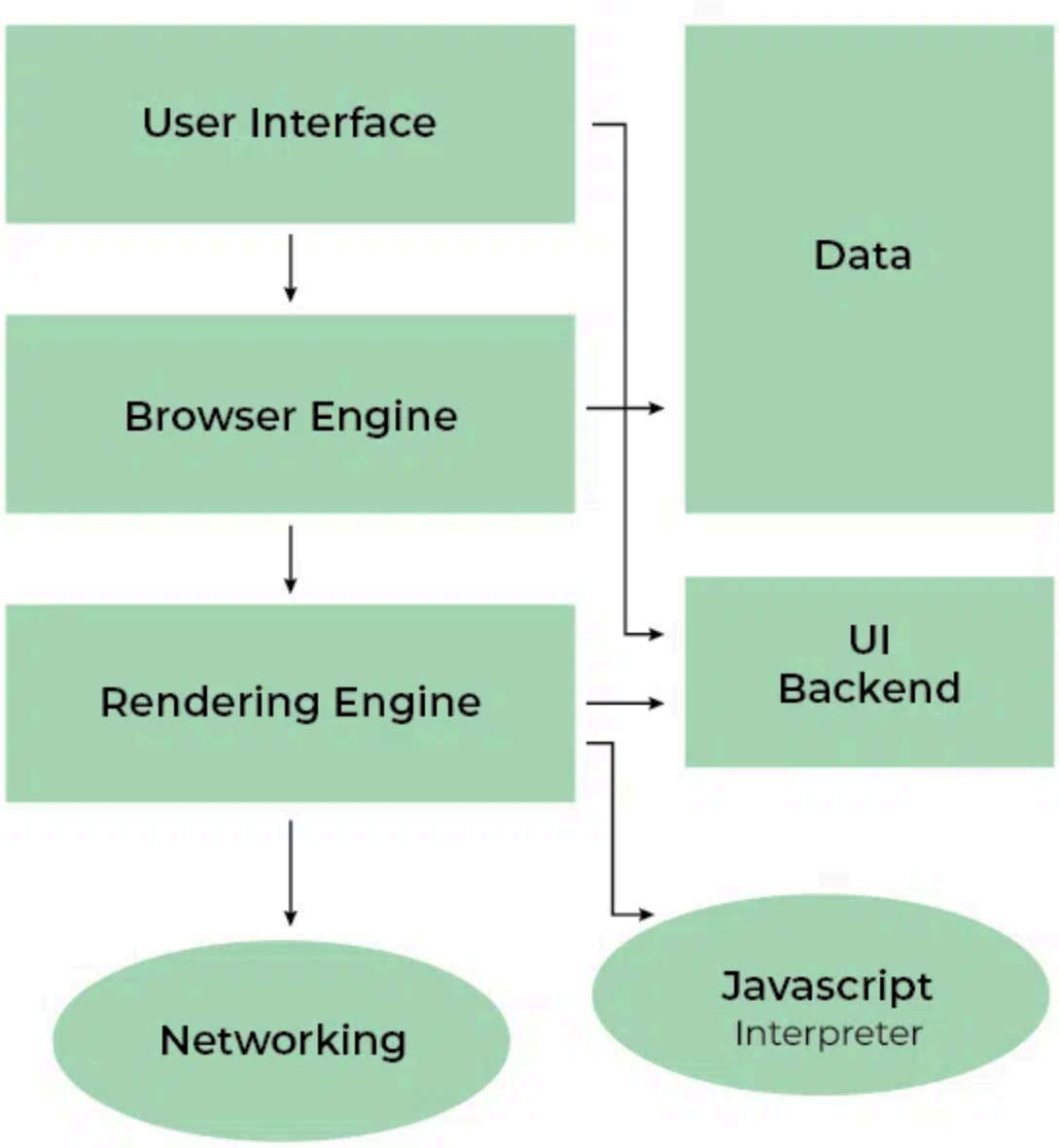
7. **Executing JavaScript**

- The browser uses its **JavaScript engine** (e.g., V8 in Chrome) to run scripts.
- Scripts can modify the DOM, handle user events, and fetch more data (AJAX).

8. **Displaying to User**

- The final rendered page is displayed to the user.
- The browser continues to listen for user interactions (clicks, scrolls, form inputs).

Web Brower Architecture:



Q5. [10 Marks] - Answers

a. What is Express JS. Explain features of Express JS.

What is Express.js?

- **Express.js** is a fast, minimalist, and flexible **web application framework** for **Node.js**.
- It simplifies the process of building **server-side applications** and **RESTful APIs**.
- Built on top of Node.js's HTTP module, it provides powerful tools for routing, middleware, and request handling.

Key Features of Express.js

1. **Routing**

- Allows defining URL paths and HTTP methods (`GET`, `POST`, `PUT`, `DELETE`) to handle client requests.
- Example:

```
app.get('/home', (req, res) => {
  res.send('Welcome Home');
});
```

2. **Middleware Support**

- Middleware functions can process requests before they reach the route handler.
- Useful for logging, authentication, parsing JSON, etc.

3. **Simplified Server Setup**

- Easy to create and configure a server with minimal code.

```
const express = require('express');
const app = express();
app.listen(3000);
```

4. **Error Handling**

- Provides structured error-handling mechanisms using middleware.

5. **Scalability and Modularity**

- Supports modular routing and controller separation, making large applications easier to manage.

6. **Compatible with REST APIs**

- Ideal for building RESTful services with clean and maintainable code.

b. Explain Flux architecture in detail.

What is Flux Architecture?

- **Flux** is a **design pattern** (not a framework) used for **managing data flow** in client-side web applications.
- It was introduced by **Facebook** to solve complexity issues in large applications, especially those using **React**.
- Its main goal is to ensure **predictable, one-way data flow**, making the app easier to debug and maintain.

The Flux Data Flow

Flux architecture is built around **four main parts**:

1. Actions

- Actions are **plain JavaScript objects** that describe *what happened* (an event).
- They **do not contain logic**, they just tell the system *something occurred* (e.g., user clicked a button).

2. Dispatcher

- The **Dispatcher** acts as the **central hub** that manages *all data flow*.
- It **receives actions** and **forwards them** to the appropriate **Stores**.
- Ensures that **every Store receives every Action**, maintaining a consistent update cycle.

---

3. Stores

- Stores hold the **application's state and logic**.
- They are **not** the same as databases.
- When a Store receives an Action from the Dispatcher:
  - It updates its data accordingly.
  - Emits a **change event** to notify the Views (React components).

4. View (React Components)

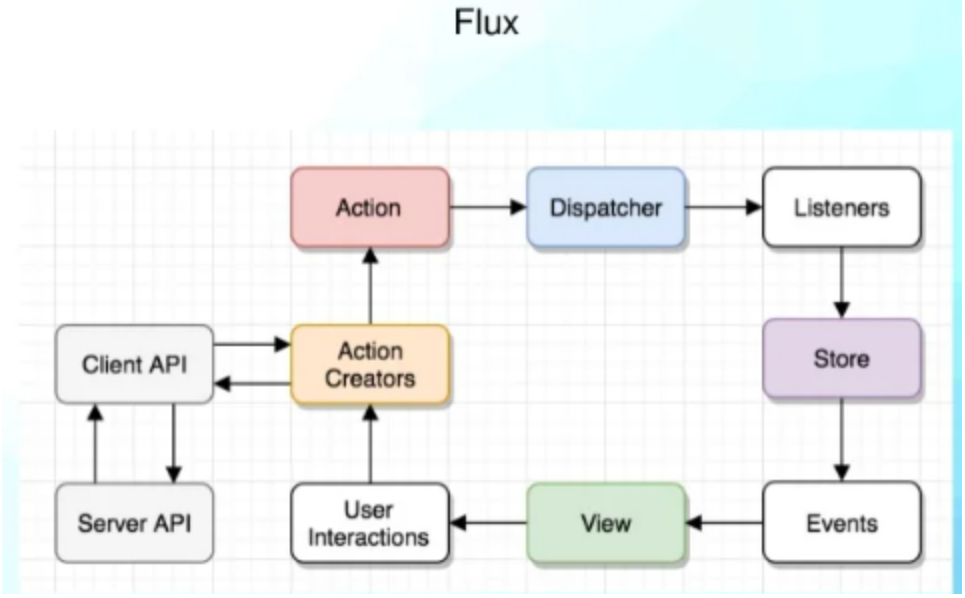
Views (React components) **listen to Stores** for data updates.

When data changes, Views **re-render** to show the new state.

They can also **trigger Actions** (via user interactions).

Unidirectional Data Flow Diagram

Here's how Flux data moves:



Data Flow in Flux (Unidirectional)

1. **User interacts** with the UI (View).
2. **Action** is created and sent to the Dispatcher.
3. **Dispatcher** forwards the action to relevant Stores.
4. **Stores** update their state and emit change events.
5. **View** listens to store changes and updates the UI.

Example Scenario: Adding a Todo

1. User clicks "Add Todo" → Action `{ type: 'ADD_TODO', payload: 'Learn Flux' }`
2. Dispatcher sends this action to the TodoStore.
3. TodoStore adds the new item and emits a change.
4. React component re-renders with the updated list.

Q6. [20 Marks] - Answers

Write a short note on any 4:

- ES5 vs ES6,
- JSX
- JSON
- URL vs URI
- Arrow Function

1. ES5 vs ES6

Introduction

ES5 (ECMAScript 5) and ES6 (ECMAScript 2015) are versions of JavaScript that introduced significant improvements in syntax, performance, and functionality.

Feature	ES5	ES6
Released Year	2009	2015
Variable Declaration	Uses `var` (function-scoped)	Introduces `let` and `const` (block-scoped)
Functions	Traditional function syntax only	Introduces **arrow functions** for shorter syntax
Classes & Modules	No class or module support	Adds `class` syntax and `import/export` modules
String Handling	Concatenation using `+`	Template literals using backticks (`` ` ``)
Default Parameters	Not supported	Supported (`function add(a=5, b=3) {}`)
Promises	Not available	Introduced for asynchronous operations

Example (Arrow Function in ES6):

```
// ES5
var add = function(a, b) { return a + b; }

// ES6
const add = (a, b) => a + b;ummary:
```

---

2. JSX (JavaScript XML)

Definition

JSX is a **syntax extension** for JavaScript used in **ReactJS** to describe what the UI should look like.

It allows mixing **HTML-like syntax** with **JavaScript logic** in the same file.

Key Features

- Looks like HTML but is converted to **React.createElement()** under the hood.
- Helps create **declarative UI components** easily.
- Supports **JavaScript expressions** inside `{}` brackets.
- Must return a **single parent element**.

Example

```
function Welcome() {
  const name = "Sameer";
  return <h2>Hello, {name}! Welcome to React.</h2>;
}
```

Advantages

- Improves code **readability**.
- Makes UI development **intuitive and faster**.

- Enables **compile-time error checking**.

---

3. JSON (JavaScript Object Notation)

Definition

JSON is a **lightweight data-interchange format** used for **storing and transferring data** between a client and server.

It is **language-independent** but based on **JavaScript syntax**.

Features

- Simple key–value pair structure.
- Supports data types like **string, number, array, object, boolean, null**.
- Commonly used in **APIs** and **configuration files**.

Syntax Example

```
{
  "name": "Sameer",
  "age": 21,
  "skills": ["Java", "React", "AWS"]
}
```

Usage in JavaScript

```
const obj = JSON.parse('{"name":"Sameer"}'); // Convert JSON → Object
const str = JSON.stringify(obj);           // Convert Object → JSON
```

---

4. URL vs URI

Definition

Both **URL** (Uniform Resource Locator) and **URI** (Uniform Resource Identifier) are used to identify resources on the internet.

Aspect	URI	URL
Definition	Identifies a resource **by name, location, or both**.	A specific type of URI that **locates** a resource on the web.
Scope	Broader — includes both URL and URN.	Narrower — only deals with **location**.
Components	Scheme, authority, path, query, fragment.	Scheme (protocol), domain, port, path, query, fragment.
Example	`urn:isbn:9780132350884` (identifies a book)	`https://www.example.com/page.html`
Purpose	Identifies **what** the resource is.	Specifies **where** the resource is and **how** to access it.

---

5. Arrow Function

Definition

Arrow functions, introduced in **ES6**, provide a **concise syntax** for writing functions in JavaScript.

They are often used for **callbacks**, **array methods**, and **React components**.

Syntax

```
// Traditional Function
function add(a, b) {
  return a + b;
}

// Arrow Function
const add = (a, b) => a + b;
```

Key Features

- **Shorter syntax** with implicit `return` for single-line expressions.
- Does **not bind its own** `this` — inherits from the parent scope.
- Cannot be used as constructors.
- Useful in **React components and array functions** like `map()`, `filter()`, etc.

Example in React

```
const Greet = () => <h2>Hello from Arrow Function!</h2>;
```