**Design Patterns**

Design patterns are common solutions to recurring problems in software design. They represent best practices and can be used to solve specific problems in a consistent and efficient manner. Here are a few common design patterns:

**Singleton Pattern**

Ensures a class has only one instance and provides a global point of access to it.

Example: Database connection

**Observer Pattern**

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

**Workflow models and types**

Workflow models represent the sequence and conditions under which tasks or activities are executed in a business process. They help in understanding, managing, and improving business processes. Here are some common workflow models and types:

Types of Workflow Models

**Sequential Workflow**

Tasks are completed in a linear sequence, one after another.

**Example**: Processing an order in an e-commerce system (Order Received -> Order Processed -> Order Shipped -> Order Delivered).

**Parallel Workflow**

Multiple tasks are executed simultaneously, and they converge at a point.

**Example**: In a project, different teams work on separate modules simultaneously, then integrate their work.

**Conditional Workflow**

The path of execution is determined by conditions or decisions.

**Example**: Loan approval process (Check Eligibility -> If Eligible, Process Loan -> If Not, Reject Loan).

**Virtual DOM**

The Virtual DOM is a lightweight, in-memory representation of the actual DOM. It is a virtual copy of the real DOM and is used to optimize updates to the UI. When changes occur, the virtual DOM is updated first. Then, the differences between the virtual DOM and the real DOM are calculated, and only the parts of the actual DOM that have changed are updated.

**MYSQL Injections**

MySQL Injection is a type of security vulnerability that occurs when an attacker is able to insert or "inject" malicious SQL code into a query. This can allow the attacker to manipulate the database, retrieve sensitive data, or even gain administrative control over the database server.

**// Example of vulnerable code**

const username = req.body.username;

const password = req.body.password;

const query = `SELECT \* FROM users WHERE username = '${username}' AND password = '${password}'`;

db.query(query, (err, result) => {

if (err) throw err;

// handle result

});

SELECT \* FROM users WHERE username = '' OR 1=1;

Since 1=1 is always true, this query returns all users, potentially granting access to the attacker.

**Preventing MySQL Injection**

This is the most effective way to prevent SQL injection. Prepared statements ensure that user input is treated as data and not executable code.

**Example in Node.js with MySQL:**

const username = req.body.username;

const password = req.body.password;

const query = 'SELECT \* FROM users WHERE username = ? AND password = ?';

db.query(query, [username, password], (err, result) => {

if (err) throw err;

// handle result

});

MySQL Injection is a serious security vulnerability that can be prevented by using prepared statements, stored procedures, escaping user inputs, using ORM libraries, and performing input validation and sanitization. Always ensure that user inputs are handled safely to protect your application and data from malicious attacks.

**Data Sanitization**

Data sanitization is the process of cleaning and validating user input to ensure that it is safe and conforms to the expected format. It is a crucial practice in web development to prevent security vulnerabilities such as SQL injection, cross-site scripting (XSS), and other attacks that exploit improperly handled user input.

Why Data Sanitization is Important

**Security**: Prevents malicious code from being executed, which can lead to data breaches and system compromise.

**Data Integrity**: Ensures that the data stored in the database is accurate and consistent.

**Reliability**: Reduces the risk of errors and unexpected behavior in applications.

**Scrum Methodology**

Scrum is an Agile framework used for managing and completing complex projects. It is particularly popular in software development but can be applied to any field. Scrum emphasizes iterative progress, collaboration, and flexibility to adapt to changing requirements.

**Key Concepts in Scrum**

**Roles:**

**Product Owner**: Defines the features of the product and prioritizes the backlog.

**Scrum Master**: Facilitates the Scrum process, helps remove impediments, and ensures that the team follows Scrum practices.

**Development Team**: A cross-functional group responsible for delivering potentially shippable product increments at the end of each sprint.

What are the 5 phases of Scrum methodology?

It is divided into five distinct phases

* **initiation**
* **planning and estimates,**
* **implementation**
* **review and retrospective**
* **release phase**.

**JS Hoisting (which data types are hoisted or which are not)**

In JavaScript, hoisting is a behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase. This allows you to use variables and functions before they are declared. However, it's important to understand that only the declarations themselves are hoisted, not their initializations or assignments.

**Variables and Hoisting**

Variables declared with var, let, and const behave differently with respect to hoisting:

**var:**

Variables declared with var are hoisted to the top of their scope (global or function scope).

However, only the declaration is hoisted, not the initialization. If you try to access a var variable before it is assigned a value, it will return undefined.

console.log(myVar); // undefined

var myVar = "Hello";  
  
**let and const:**

Variables declared with let and const are hoisted to the top of their block scope (like var), but they are not initialized until their declaration is evaluated.

console.log(myLet); // ReferenceError: Cannot access 'myLet' before initialization

let myLet = "World";  
  
This is because let and const are in the "temporal dead zone" (TDZ) until their actual declaration is encountered during execution.

**Temporal Dead Zone (TDZ)**

The TDZ is the period between the start of the scope (where the variable is declared) and the initialization point (where the variable is assigned a value). During this phase, any attempt to access the variable will result in a ReferenceError.

**Example of Temporal Dead Zone**

console.log(myVar); // undefined

var myVar = "Hello";

console.log(myLet); // ReferenceError: Cannot access 'myLet' before initialization

let myLet = "World";

**In this example:**

myVar is declared with var, so it's hoisted to the top of its scope. The console.log statement shows undefined because initialization (assignment) happens later.

myLet is declared with let, so it's also hoisted to the top of its scope. However, since initialization hasn't occurred yet, accessing myLet before its declaration throws a ReferenceError.

**Benefits of TDZ**

**Error Prevention**: Helps catch potential issues where variables are accessed before they are initialized, which can lead to bugs.

**Predictability**: Encourages cleaner code by enforcing a clear separation between declaration and initialization points.

**Strictness**: Promotes better coding practices by emphasizing the importance of understanding variable lifecycles.

**Callback Function & Callback Hell**

A callback function is a function that is passed as an argument to another function and is executed after some operation has been completed. Callback functions are commonly used for handling asynchronous operations in JavaScript, such as reading files, making network requests, or setting timers.

**Example of a Callback Function**

function greeting(name) {

console.log("Hello " + name);

}

function processUserInput(callback) {

const name = prompt("Please enter your name.");

callback(name);

}

processUserInput(greeting);

**Callback Hell**

Callback Hell (also known as the "Pyramid of Doom") refers to a situation where multiple nested callbacks are used to perform asynchronous operations. This can lead to code that is difficult to read, understand, and maintain due to deep nesting and complex control flow.

**Example of Callback Hell**

getData(function(err, data) {

if (err) {

console.error(err);

} else {

processData(data, function(err, processedData) {

if (err) {

console.error(err);

} else {

writeFile(processedData, function(err) {

if (err) {

console.error(err);

} else {

console.log("Success!");

}

});

}

});

}

});

**Avoiding Callback Hell**

* Use Promises
* **Async/Await**

**Lexical Scope**

Lexical scope (or static scope) in JavaScript refers to the scope that is determined at compile time, based on the location of the variables, functions, and blocks within the written code. The lexical scope defines how variable names are resolved in nested functions: inner functions contain the scope of parent functions even if the parent function has returned.

**Example of Lexical Scope**

function outerFunction() {

const outerVariable = "I am outside!";

function innerFunction() {

const innerVariable = "I am inside!";

console.log(outerVariable); // Accessing outer variable

console.log(innerVariable); // Accessing inner variable

}

innerFunction();

// console.log(innerVariable); // ReferenceError: innerVariable is not defined

}

outerFunction();

**Lexical Scope in Closures**

function createCounter() {

let count = 0;

return function() {

count++;

console.log(count);

};

}

const counter = createCounter();

counter(); // 1

counter(); // 2

counter(); // 3

**In this example:**

* createCounter returns an anonymous function that increments and logs count.
* The anonymous function forms a closure, retaining access to count even after createCounter has finished executing.
* Each call to counter increments and logs the updated count because the closure maintains its reference to the lexical scope where count was defined.

Lexical scope is a fundamental concept in JavaScript that influences how variables are accessed and managed in nested functions and closures.

**Higher-Order Components (HOCs)**

Higher-Order Components (HOCs) are a pattern in React for reusing component logic. An HOC is a function that takes a component and returns a new component, effectively wrapping the original component with additional functionality.

Practical Use Cases for HOCs

**Authorization**

Use HOCs to restrict access to certain components based on user roles or permissions.

const withAuthorization = (WrappedComponent) => {

return (props) => {

if (!props.isAuthenticated) {

return <Redirect to="/login" />;

}

return <WrappedComponent {...props} />;

};

};

**Fetching Data**:

Use HOCs to fetch data from an API and pass it as props to the wrapped component.

const withData = (url) => (WrappedComponent) => {

return class extends React.Component {

state = { data: null };

componentDidMount() {

fetch(url)

.then((response) => response.json())

.then((data) => this.setState({ data }));

}

render() {

return <WrappedComponent {...this.props} data={this.state.data} />;

}

};

};

**Anonymous Functions**

Anonymous functions are functions that are defined without a name. They are often used for short-term tasks, such as passing a function as an argument to another function or immediately invoking a function.

**Characteristics of Anonymous Functions**

**No Name:**

* They do not have a name identifier.
* This makes them useful for quick, one-time use but harder to reference in debugging.

**Assigned to Variables:**

* They can be assigned to variables or constants.
* This allows them to be reused through the variable name.

**Used as Arguments:**

* Commonly used as arguments in higher-order functions (e.g., map, filter, reduce).

**Immediate Invocation:**

* Often used in Immediately Invoked Function Expressions (IIFEs).

**Example of an Anonymous Function**

**Assigned to a Variable**:

const greet = function(name) {

return `Hello, ${name}!`;

};

console.log(greet("Alice")); // Output: "Hello, Alice!"

**Used as a Callback**:

setTimeout(function() {

console.log("This runs after 2 seconds");

}, 2000);

**IIFE (Immediately Invoked Function Expression)**:

(function() {

console.log("This runs immediately!");

})();

**Debouncing**

Debouncing is a programming technique used to ensure that a function is only executed once after a specified delay period, even if it is called multiple times within that period. This is particularly useful for performance optimization in scenarios where an event can fire multiple times in quick succession, such as window resizing, key presses, or scrolling.

**Key Concepts of Debouncing**

* **Delay**: The function will only execute after a specified delay period has passed since the last time it was invoked.
* **Optimization**: Reduces the number of times a function is called, which can improve performance by preventing unnecessary processing.

Example of Debouncing

Consider an input field where a search query is sent to a server each time the user types a character. Without debouncing, this can result in a flood of network requests. With debouncing, the search function is called only after the user has stopped typing for a certain period.

Implementation of Debouncing

**Here's a simple implementation of a debounce function in JavaScript:**

const searchInput = document.getElementById('searchInput');

const handleSearch = debounce((event) => {

console.log('Search query:', event.target.value);

// Perform the search operation here

}, 300);

searchInput.addEventListener('input', handleSearch);

**In this example:**

* The debounce function takes two arguments: the function to debounce (func) and the delay in milliseconds (delay).x`
* The handleSearch function is created by debouncing the actual search function with a 300ms delay.
* The input event listener on the search input field calls handleSearch, ensuring that the search function is only invoked after the user stops typing for 300 milliseconds.

**Summary**

* **Debouncing**: A technique to ensure a function is executed only once after a specified delay period, even if invoked multiple times.
* **Purpose**: Optimizes performance by reducing the number of function calls.

**Thread Pooling**

Thread pooling is a technique used in concurrent programming to manage a collection of reusable threads for executing tasks. Instead of creating and destroying threads for each task, which can be resource-intensive, a pool of threads is maintained and reused. This approach improves the efficiency and performance of multi-threaded applications by reducing the overhead associated with thread creation and destruction.

**Summary**

* **Thread Pooling**: A technique to manage a collection of reusable threads for executing tasks, improving performance and resource management.
* **Benefits**: Performance improvement, efficient resource management, scalability, and simplified code.

**JS Array Methods**

**Array.from()**

The Array.from() method in JavaScript creates a new, shallow-copied Array instance from an array-like or iterable object. This method is useful for converting structures that are not true arrays, like NodeLists, strings, or sets, into actual arrays.

**const str = 'hello';**

**const array = Array.from(str);**

**console.log(array); // ['h', 'e', 'l', 'l', 'o']**

**Array.of()**

It provides a way to create arrays from a list of elements, unlike the Array constructor which can behave differently based on the number of arguments provided.

**const array = Array.of(1, 2, 3, 4, 5);**

**console.log(array); // [1, 2, 3, 4, 5]**

**pop()**

The pop() method in JavaScript is used to remove the last element from an array and return that element. This method changes the length of the array.

**let fruits = ['apple', 'banana', 'cherry'];**

**let lastFruit = fruits.pop();**

**console.log(lastFruit); // Output: "cherry"**

**console.log(fruits); // Output: ["apple", "banana"]**

**filter()**

The filter() method in JavaScript creates a new array with all elements that pass the test implemented by the provided function. It does not modify the original array.

**let students = [**

**{ name: 'Alice', grade: 85 },**

**{ name: 'Bob', grade: 92 },**

**{ name: 'Charlie', grade: 78 },**

**{ name: 'David', grade: 90 }**

**];**

**let topStudents = students.filter(student => student.grade > 80);**

**console.log(topStudents);**

**// Output:**

**// [**

**// { name: 'Alice', grade: 85 },**

**// { name: 'Bob', grade: 92 },**

**// { name: 'David', grade: 90 }**

**// ]**

**Find()**

The find() method in JavaScript returns the value of the first element in the array that satisfies the provided testing function. If no elements satisfy the testing function, undefined is returned.

**let students = [**

**{ name: 'Alice', grade: 85 },**

**{ name: 'Bob', grade: 92 },**

**{ name: 'Charlie', grade: 78 },**

**{ name: 'David', grade: 90 }**

**];**

**let topStudent = students.find(student => student.grade > 90);**

**console.log(topStudent); // Output: { name: 'Bob', grade: 92 }**

**reduceRight()**

The reduceRight() method in JavaScript applies a function against an accumulator and each element in the array (from right to left) to reduce it to a single value. It iterates over the elements of the array in reverse order.

**let numbers = [1, 2, 3, 4, 5];**

**let sum = numbers.reduceRight((accumulator, currentValue) => accumulator + currentValue);**

**console.log(sum); // Output: 15 (5 + 4 + 3 + 2 + 1)**

**Every()**

The every() method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

**let numbers = [1, 2, 3, 4, 5];**

**let allPositive = numbers.every(number => number > 0);**

**console.log(allPositive); // Output: true**

**Some()**

The some() method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

**let numbers = [1, 2, 3, 4, -5];**

**let anyNegative = numbers.some(number => number < 0);**

**console.log(anyNegative); // Output: true**

**includes()**

The includes() method determines whether an array includes a certain value among its elements, returning true or false as appropriate.

**let numbers = [1, 2, 3, 4, 5];**

**console.log(numbers.includes(3)); // Output: true**

**console.log(numbers.includes(8)); // Output: false**

**lastIndexOf()**

The lastIndexOf() method in JavaScript returns the last index at which a given element can be found in the array, or -1 if the element is not present. It searches the array from right to left.

**let numbers = [1, 2, 3, 4, 2, 5, 2];**

**console.log(numbers.lastIndexOf(2)); // Output: 6 (last index of number 2)**

**console.log(numbers.lastIndexOf(6)); // Output: -1 (element 6 is not found)**

**concat()**

The concat() method is used to merge arrays. It does not change the existing arrays; instead, it returns a new array.

**let arr1 = [1, 2, 3];**

**let arr2 = [4, 5];**

**let arr3 = [6, 7, 8];**

**let concatenatedArray = arr1.concat(arr2, arr3);**

**console.log(concatenatedArray); // Output: [1, 2, 3, 4, 5, 6, 7, 8]**

**join()**

The join() method converts all elements of an array into a string and concatenates them, optionally separating them with a specified separator string.

**let fruits = ['Apple', 'Banana', 'Orange'];**

**let joinedString = fruits.join(' - ');**

**console.log(joinedString); // Output: "Apple - Banana - Orange"**

**flat()**

The flat() method returns a new array with all sub-array elements concatenated into it recursively up to the specified depth.

**let arr = [1, 2, [3, 4]];**

**let flattenedArray = arr.flat();**

**console.log(flattenedArray); // Output: [1, 2, 3, 4]**

**flatMap()**

The flatMap() method returns a new array formed by applying a mapping function to each element of the array, and then flattening the result into a new array.

**let arr = [1, 2, 3];**

**let mappedArray = arr.flatMap(x => [x \* 2]);**

**console.log(mappedArray); // Output: [2, 4, 6]**

**reverse()**

The reverse() method in JavaScript reverses the order of elements in an array in place.

**let numbers = [1, 2, 3, 4, 5];**

**numbers.reverse();**

**console.log(numbers); // Output: [5, 4, 3, 2, 1]**

**How to solve Callback Hell**

Callback Hell, also known as the Pyramid of Doom, is a situation where callbacks are nested within other callbacks multiple levels deep, making the code hard to read and maintain. There are several ways to solve Callback Hell:

* **Named Functions**: Breaks down the code into smaller, more manageable pieces, improving readability.
* **Promises**: Allows chaining of asynchronous operations and provides better error handling with .then() and .catch().
* **Async/Await**: Makes asynchronous code look like synchronous code, further improving readability and maintainability.

Using these methods, you can avoid Callback Hell and write cleaner, more maintainable asynchronous JavaScript code.

**Difference between promise and async/await**

Promises and async/await are both used to handle asynchronous operations in JavaScript, but they provide different syntactical approaches and some differences in functionality and use cases. Here's a comparison of both:

**Promises**

A Promise represents a value that may be available now, or in the future, or never. Promises provide a way to handle asynchronous operations in a more manageable way compared to callbacks.  
  
**Async/Await**

async/await is syntactic sugar built on top of promises. It makes asynchronous code look and behave more like synchronous code, using async functions and the await keyword.

**Key Differences**

**Readability and Syntactic Sugar:**

Promises: Require chaining with .then() and .catch(), which can become cumbersome with complex operations.

Async/Await: Provides a more readable and cleaner syntax that looks like synchronous code, improving readability.

**Error Handling:**

Promises: Use .catch() for handling errors.

Async/Await: Use try/catch blocks for handling errors, making the error handling syntax more consistent with synchronous code.

**Sequential vs Parallel Execution:**

Promises: Promises can be used for parallel execution with Promise.all(), Promise.allSettled(), etc.

Async/Await: By default, await waits for the promise to resolve before moving to the next line, making it easier to write sequential asynchronous code. For parallel execution, you still use Promise.all() within an async function.

**Sequentail for await/async**

async function sequentialFetch() {

const result1 = await fetchData1();

const result2 = await fetchData2();

return [result1, result2];

}

**Parallel with Async/Await**  
async function parallelFetch() {

const promise1 = fetchData1();

const promise2 = fetchData2();

const results = await Promise.all([promise1, promise2]);

return results;

}

**Is JS single thread or multi thread,?**

JavaScript is single-threaded. This means it executes code in a single sequence, one command at a time. However, it can handle asynchronous operations using event loops, callbacks, promises, and async/await.  
  
console.log("Start");

// Simulate an asynchronous operation using setTimeout

setTimeout(() => {

console.log("This runs after 2 seconds");

}, 2000);

console.log("End");  
  
**In this example:**

* "Start" is printed.
* The setTimeout function schedules a task to run after 2 seconds and immediately moves to the next line.
* "End" is printed.
* After 2 seconds, "This runs after 2 seconds" is printed.

**Promises (resolve, reject)**

A Promise in JavaScript is an object representing the eventual completion or failure of an asynchronous operation. Promises provide a more elegant way to handle asynchronous operations compared to traditional callback methods.

A Promise is created using the Promise constructor, which takes a function (executor) with two parameters: resolve and reject.  
  
let promise = new Promise((resolve, reject) => {

// asynchronous operation

if (/\* operation successful \*/) {

resolve(value);

} else {

reject(error);

}

});

**States of a Promise**

* **Pending**: Initial state, neither fulfilled nor rejected.
* **Fulfilled**: The operation completed successfully, resolve was called.
* **Rejected**: The operation failed, reject was called.

**How Virtual DOM Works**

The Virtual DOM is a concept used in libraries like React to improve the performance and efficiency of web applications. It is a lightweight copy of the real DOM that allows libraries to update the UI more efficiently. Here's how it works:

* When a component is first rendered, the virtual DOM creates a virtual representation of the real DOM elements.
* When the state of a component changes, the virtual DOM updates its representation to reflect these changes.
* The virtual DOM compares the new virtual DOM with the previous virtual DOM to determine what has changed. This process is known as "diffing".
* After identifying the differences, the virtual DOM updates only the parts of the real DOM that have changed, minimizing the number of manipulations and reflows.

**State vs Prop**

**State**

State is a built-in object that is used to store property values that belong to a component. When the state object changes, the component re-renders to reflect those changes. State is local to the component and can be changed within the component.

* **Mutable**: Can change within the component.
* **Local**: Managed and used within the component.
* **Example** **Usage**: Managing user input, toggling UI elements.

**Props**

Props (short for properties) are read-only attributes used to pass data from one component to another, typically from a parent to a child. Props are immutable, meaning they cannot be modified by the receiving component.

* **Immutable**: Read-only and cannot be changed by the child component.
* **Passed from Parent to Child**: Used to pass data and configuration down the component tree.
* **Example Usage**: Passing data to display, configuring child components.

**Data Binding**

Data binding in web development refers to the technique of synchronizing data between the UI (User Interface) and the business logic. It allows the automatic synchronization of data between the model (data) and the view (UI elements). This can be achieved in various ways, depending on the framework or library you are using. Here are some common approaches and examples:  
  
**One-Way Data Binding**

In one-way data binding, the data flow is unidirectional, from the model to the view. Changes in the model automatically update the view.

**Two-Way Data Binding**

In two-way data binding, the data flow is bidirectional. Changes in the model update the view, and changes in the view update the model.  
  
**Data Binding in React**

React uses a one-way data flow but can achieve similar functionality to two-way data binding through controlled components and state management.

**import React, { useState } from 'react';**

**function App() {**

**const [message, setMessage] = useState("Hello, World!");**

**return (**

**<div>**

**<input**

**type="text"**

**value={message}**

**onChange={(e) => setMessage(e.target.value)}**

**/>**

**<p>{message}</p>**

**</div>**

**);**

**}**

**export default App;**

**React Hooks (useState, useEffect, useMemo, useCallback, useLayoutEffect, useContext)**

**useState**

useState is used to add state to functional components.

const [state, setState] = useState(initialState);

**useEffect**

useEffect is used to handle side effects in functional components, such as fetching data, setting up subscriptions, or manually changing the DOM.

**useEffect(() => {**

**// Effect code here**

**return () => {**

**// Cleanup code here (optional)**

**};**

**}, [dependencies]);**

**useMemo**

useMemo memoizes a computation, recomputing it only when its dependencies change. It's useful for optimizing performance.

**import React, { useState, useCallback } from 'react';**

**function Button({ onClick }) {**

**return <button onClick={onClick}>Click me</button>;**

**}**

**function ParentComponent() {**

**const [count, setCount] = useState(0);**

**const handleClick = useCallback(() => {**

**setCount(count + 1);**

**}, [count]);**

**return <Button onClick={handleClick} />;**

**}**

**useContext**

useContext is used to access context values in a functional component. It simplifies context consumption without needing a render prop or Consumer component.

**import React, { useContext } from 'react';**

**const ThemeContext = React.createContext('light');**

**function ThemedComponent() {**

**const theme = useContext(ThemeContext);**

**return <div className={theme}>This is a {theme} themed component.</div>;**

**}**

**function App() {**

**return (**

**<ThemeContext.Provider value="dark">**

**<ThemedComponent />**

**</ThemeContext.Provider>**

**);**

**}**

**Summary**

**useState**: Adds state to functional components.

**useEffect**: Manages side effects (e.g., data fetching, subscriptions).

**useMemo**: Memoizes computations for performance optimization.

**useCallback**: Memoizes callback functions to prevent unnecessary re-renders.

**useLayoutEffect**: Similar to useEffect, but fires synchronously after DOM mutations.

**useContext**: Accesses context values without using a Consumer component.

**Difference Between useEffect and useLayoutEffect**

**useEffect**:

* Runs after the render and paint.
* Asynchronous and non-blocking.
* Ideal for non-urgent side effects like data fetching.

**useLayoutEffect**:

* Runs synchronously after DOM updates but before the paint.
* Blocking and can affect performance if not used carefully.
* Ideal for DOM measurements and updates that need to be completed before paint.

**React Workflows: Flux and Redux**

React workflows are architectural patterns that help manage the state and data flow in React applications. Two of the most commonly used workflows are Flux and Redux. Both aim to make state management more predictable and maintainable, but they have different implementations and use cases.

**Flux**

Flux is an architecture for building client-side web applications. It emphasizes unidirectional data flow, making it easier to reason about changes to the application's state.

**Key Concepts**

**Actions:**

Actions are plain JavaScript objects that contain a type field and any additional data needed to describe what happened. They are the only source of information for the store.

**Example:**

const addItemAction = {

type: 'ADD\_ITEM',

item: 'NewItem'

};

**Dispatcher**:

The dispatcher is a central hub that manages all actions. It dispatches actions to the stores.

**Example:**

import { Dispatcher } from 'flux';

const dispatcher = new Dispatcher();

dispatcher.dispatch(addItemAction);

**Stores:**

Stores hold the application's state and logic. Each store registers with the dispatcher and provides a callback that the dispatcher can call when an action is dispatched.

**Example**:

class ItemStore {

constructor() {

this.items = [];

this.dispatchToken = dispatcher.register(this.handleAction.bind(this));

}

handleAction(action) {

switch (action.type) {

case 'ADD\_ITEM':

this.items.push(action.item);

this.emitChange();

break;

}

}

emitChange() {

// Notify change to the view

}

}

**Views**:

Views are React components that listen to changes in stores and re-render in response to these changes.

**Example**:

class ItemListView extends React.Component {

componentDidMount() {

itemStore.on('change', this.handleStoreChange);

}

handleStoreChange = () => {

this.setState({ items: itemStore.items });

};

render() {

return (

<ul>

{this.state.items.map(item => <li key={item}>{item}</li>)}

</ul>

);

}

}

**Redux**

Redux is a predictable state container for JavaScript apps, often used with React. It builds on the ideas of Flux but with a simplified and more centralized approach.

**Key Concepts**

**Actions:**

Actions are plain objects that represent an intention to change the state.

**Example**:

const addItemAction = {

type: 'ADD\_ITEM',

item: 'NewItem'

};

**Reducers**:

Reducers are pure functions that take the current state and an action, and return a new state.

**Example:**

function itemsReducer(state = [], action) {

switch (action.type) {

case 'ADD\_ITEM':

return [...state, action.item];

default:

return state;

}

}

**Store:**

The store holds the state of the application. It is created using the createStore function, which takes a root reducer and an initial state.

**Example:**

import { createStore } from 'redux';

const store = createStore(itemsReducer);

store.dispatch(addItemAction);

console.log(store.getState()); // ['NewItem']

**Provider**:

The Provider component makes the Redux store available to the rest of your app.

**Example:**

**import { Provider } from 'react-redux';**

import { createStore } from 'redux';

import itemsReducer from './reducers';

import App from './App';

const store = createStore(itemsReducer);

function Root() {

return (

<Provider store={store}>

<App />

</Provider>

);

}

**Connect**:

The connect function connects React components to the Redux store.

**Example:**

import { connect } from 'react-redux';

const mapStateToProps = state => ({

items: state.items

});

const mapDispatchToProps = dispatch => ({

addItem: item => dispatch({ type: 'ADD\_ITEM', item })

});

const ItemList = ({ items, addItem }) => (

<div>

<ul>

{items.map((item, index) => <li key={index}>{item}</li>)}

</ul>

<button onClick={() => addItem('NewItem')}>Add Item</button>

</div>

);

export default connect(mapStateToProps, mapDispatchToProps)(ItemList);

**Summary**

**Flux**:

* Unidirectional data flow.
* Actions, dispatcher, stores, and views.
* More boilerplate and less centralized.

**Redux:**

* Predictable state container.
* Actions, reducers, and store.
* Simplified with a more centralized approach.

**Difference Between localStorage and sessionStorage**

**localStorage**

localStorage is used for storing data that needs to persist across browser sessions, meaning the data is not cleared when the browser is closed.

***Key Characteristics***

**Persistence:**

Data stored in localStorage remains until explicitly deleted by the user or the application.

It persists even after the browser is closed and reopened.

**Scope:**

localStorage is scoped to the origin (domain). Data stored can be accessed by all pages of the same origin.

**Storage Capacity:**

Typically, localStorage can store up to 5-10 MB of data, but this can vary depending on the browser.

**Usage:**

Used for storing data that needs to persist long-term, such as user preferences, tokens, or application state.

Example:

// Store data

localStorage.setItem('key', 'value');

// Retrieve data

const value = localStorage.getItem('key');

console.log(value); // 'value'

// Remove data

localStorage.removeItem('key');

// Clear all data

localStorage.clear();