**Answer 1**

**1 ) Storage Scope:**

Cookies: Sent to the server with every HTTP request and can be used for both client-side and server-side storage.

Local Storage: Stored on the client-side and persists even after the browser is closed. It has a domain-level scope and is accessible across different tabs and windows of the same domain.

Session Storage: Also stored on the client-side, but its data is limited to the current session. It is accessible within the same tab or window until the tab or window is closed.

**2 ) Storage Limit:**

Cookies: Limited to 4KB in size.

Local Storage: Larger storage capacity, typically around 5-10MB depending on the browser.

Session Storage: Similar storage capacity to local storage, around 5-10MB.

**3) Accessibility:**

Cookies: Can be accessed on both the client-side (JavaScript) and server-side (server languages like PHP, Node.js, etc.).

Local Storage: Accessible only on the client-side through JavaScript.

Session Storage: Accessible only on the client-side through JavaScript.

**Answer 2**

**So, the output will be:**

**5**

**5**

**5**

**5**

**5**

Each 5 will be printed after a 100-millisecond delay between each log statement, but they all log the value of i after the loop finishes, which is 5.

**Answer 3**

Sharding in MongoDB is a technique used to distribute data across multiple servers or nodes to improve scalability and performance. It allows MongoDB to handle large datasets and high read/write workloads efficiently.

In short, sharding works as follows:

1 Data Distribution: The data in a MongoDB database is divided into smaller chunks called shards. Each shard represents a subset of the data. The distribution can be based on a shard key, which is a field or combination of fields chosen to determine the data's placement in the shards.

2 Shard Cluster: A MongoDB shard cluster consists of multiple servers, with each server responsible for managing one or more shards. These servers are known as shard servers or shard nodes.

3 Shard Routing: A mongos process acts as a router in the application tier. It receives client requests and determines which shard(s) the request should be directed to based on the shard key.

**Answer 4**

Promise chaining is a technique in JavaScript used to handle asynchronous operations sequentially by chaining multiple promises together. It allows you to perform a series of asynchronous tasks one after another in a more readable and concise manner.

promise chaining involves attaching multiple .then()or .catch() handlers to a promise, where each handler returns another promise or a value.

Example:

function fetchUserData() {

return new Promise((resolve) => {

setTimeout(() => resolve({ name: "John", age: 30 }), 1000);

});

}

function processUserData(userData) {

return new Promise((resolve) => {

setTimeout(() => {

userData.email = "john@example.com";

resolve(userData);

}, 500);

});

}

function displayUserData(userData) {

console.log(userData);

}

// Promise chaining:

fetchUserData()

.then(processUserData)

.then(displayUserData)

.catch((error) => console.error("Error:", error));

In this example, we have three functions: **fetchUserData**, **processUserData**, and **displayUserData**. Each function returns a promise that resolves to the respective data or performs some asynchronous operations.

**Answer 5**

Higher-Order Components (HOC) in React are functions that take a component as an input and return a new enhanced component. They are a pattern used to share code, logic, and functionality between multiple components in a reusable way.

In short, HOCs work as follows:

1. **Input Component:** You start with a regular React component (the input component) that you want to enhance with additional functionality.

2. **HOC Function:** Then, you create a higher-order component function that accepts the input component as an argument.

3. **Enhanced Component:** Inside the HOC function, you can add new props, manipulate the existing ones, or provide additional functionality to the input component.

Example:

// Higher-Order Component (HOC) function

function withLogger(WrappedComponent) {

return class extends React.Component {

componentDidMount() {

console.log("Component was mounted.");

}

render() {

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

}

};

}

// Regular React component

class MyComponent extends React.Component {

render() {

return <div>My Component</div>;

}

}

// Enhance MyComponent with the withLogger HOC

const EnhancedComponent = withLogger(MyComponent);

// Usage of the enhanced component

ReactDOM.render(<EnhancedComponent />, document.getElementById("root"));

**Answer 7**

const arr = [1, 2, 3, 4, 5];

const reversedArr = arr.reduce((acc, current) => {

acc.unshift(current);

return acc;

}, []);

console.log(reversedArr);

**Answer 8**

So, the order of logs will be:

**1**

**4**

**3**

**2**

console.log(1)- will log 1 to the console first, as it is executed synchronously and immediately.

console.log(4)- will log 4 to the console after 1, as it is also executed synchronously and immediately, following the previous statement.

setTimeout(function(){console.log(3)}, 0)- will log 3 to the console after 1 and 4. Despite the timeout being set to 0, the function inside the setTimeout is not executed immediately.Since 1 and 4 are already logged, 3 is next in line to be logged.

setTimeout(function(){console.log(2)}, 1000)- will log 2 to the console last, after a delay of 1000 milliseconds (1 second). This is because it is set to execute after a timeout of 1000ms, so it will be the last one to be executed among all the logged statements.

**Answer 9**

array 1: length=5 last=j,o,n,e,s

array 2: length=5 last=j,o,n,e,s