### 1. Memoization Function

\*\*Question:\*\*

Write a function `memoize` that takes a function `fn` as an argument and returns a memoized version of `fn`. The memoized function should cache the results based on the arguments passed.

\*\*Example:\*\*

```javascript

function add(a, b) {

return a + b;

}

const memoizedAdd = memoize(add);

console.log(memoizedAdd(1, 2)); // 3 (calculated)

console.log(memoizedAdd(1, 2)); // 3 (cached)

console.log(memoizedAdd(2, 3)); // 5 (calculated)

console.log(memoizedAdd(2, 3)); // 5 (cached)

```

\*\*Expected Output:\*\*

- The function should calculate and cache results based on the inputs.

### 2. Deep Clone

\*\*Question:\*\*

Write a function `deepClone` that performs a deep clone of a given object. The function should handle nested objects and arrays.

\*\*Example:\*\*

```javascript

const original = {

a: 1,

b: { c: 2, d: [3, 4] },

e: [{ f: 5 }, { g: 6 }]

};

const cloned = deepClone(original);

console.log(cloned); // Should be a deep copy of `original`

console.log(cloned !== original); // true

console.log(cloned.b !== original.b); // true

console.log(cloned.e[0] !== original.e[0]); // true

```

\*\*Expected Output:\*\*

- The function should return a new object that is a deep clone of the input.

### 3. Event Emitter

\*\*Question:\*\*

Implement a simple `EventEmitter` class that allows for event registration,

deregistration, and emitting events.

\*\*Example:\*\*

```javascript

class EventEmitter {

// Implement methods on, off, and emit

}

const emitter = new EventEmitter();

function callback(data) {

console.log('Event received with data:', data);

}

emitter.on('event1', callback);

emitter.emit('event1', { some: 'data' }); // Event received with data: { some: 'data' }

emitter.off('event1', callback);

emitter.emit('event1', { some: 'data' }); // No output

```

\*\*Expected Output:\*\*

- The `on` method registers an event listener.

- The `off` method deregisters an event listener.

- The `emit` method triggers all listeners for a given event.

### 4. Custom Promise.all

\*\*Question:\*\*

Write a function `promiseAll` that mimics the behavior of `Promise.all`. It should take an array of promises and return a single promise that resolves with an array of results once all input promises have resolved, or rejects if any promise rejects.

\*\*Example:\*\*

```javascript

function promiseAll(promises) {

// Implement the function

}

const promise1 = Promise.resolve(3);

const promise2 = 42;

const promise3 = new Promise((resolve, reject) => {

setTimeout(resolve, 100, 'foo');

});

promiseAll([promise1, promise2, promise3]).then((results) => {

console.log(results); // [3, 42, "foo"]

});

```

\*\*Expected Output:\*\*

- The function should handle an array of promises and return a promise that resolves with an array of results.

### 5. Function Debounce

\*\*Question:\*\*

Implement a function `debounce` that returns a debounced version of the given function. The debounced function delays the execution until after `wait` milliseconds have elapsed since the last time it was invoked.

\*\*Example:\*\*

```javascript

function debounce(fn, wait) {

// Implement the function

}

const log = debounce((message) => console.log(message), 2000);

log('Hello'); // Will only log "Hello" if no other calls are made within 2000ms

```

\*\*Expected Output:\*\*

- The function should prevent the `fn` from being called until after the specified wait time has passed without another call.

### 6. Currying Function

\*\*Question:\*\*

Write a function `curry` that takes a function `fn` and returns a curried version of `fn`.

\*\*Example:\*\*

```javascript

function curry(fn) {

// Implement the function

}

function add(a, b, c) {

return a + b + c;

}

const curriedAdd = curry(add);

console.log(curriedAdd(1)(2)(3)); // 6

console.log(curriedAdd(1, 2)(3)); // 6

console.log(curriedAdd(1)(2, 3)); // 6

```

\*\*Expected Output:\*\*

- The function should allow partial application of the arguments.

### 6. To-Do List with Local Storage

\*\*Question:\*\*

Create a simple To-Do List application that allows users to add, remove, and mark tasks as complete. Use Local Storage to persist the tasks.

\*\*Requirements:\*\*

- Add tasks

- Remove tasks

- Mark tasks as complete/incomplete

- Persist tasks using Local Storage

\*\*Example:\*\*

```javascript

class TodoApp {

constructor() {

this.tasks = JSON.parse(localStorage.getItem('tasks')) || [];

this.render();

}

addTask(task) {

this.tasks.push({ task, completed: false });

this.save();

this.render();

}

removeTask(index) {

this.tasks.splice(index, 1);

this.save();

this.render();

}

toggleTask(index) {

this.tasks[index].completed = !this.tasks[index].completed;

this.save();

this.render();

}

save() {

localStorage.setItem('tasks', JSON.stringify(this.tasks));

}

render() {

const list = document.getElementById('task-list');

list.innerHTML = '';

this.tasks.forEach((task, index) => {

const li = document.createElement('li');

li.textContent = task.task;

li.style.textDecoration = task.completed ? 'line-through' : 'none';

li.onclick = () => this.toggleTask(index);

const removeButton = document.createElement('button');

removeButton.textContent = 'Remove';

removeButton.onclick = (e) => {

e.stopPropagation();

this.removeTask(index);

};

li.appendChild(removeButton);

list.appendChild(li);

});

}

}

const app = new TodoApp();

document.getElementById('add-task-button').onclick = () => {

const taskInput = document.getElementById('task-input');

if (taskInput.value) {

app.addTask(taskInput.value);

taskInput.value = '';

}

};

```

### 7. Tic-Tac-Toe Game

\*\*Question:\*\*

Create a Tic-Tac-Toe game where two players can play against each other. The game should display the current game state and declare a winner or a draw.

\*\*Requirements:\*\*

- A 3x3 grid for the game

- Two players (X and O)

- Detect win or draw

- Reset functionality

\*\*Example:\*\*

```javascript

class TicTacToe {

constructor() {

this.board = Array(9).fill(null);

this.currentPlayer = 'X';

this.render();

}

handleClick(index) {

if (this.board[index] || this.checkWinner()) return;

this.board[index] = this.currentPlayer;

if (this.checkWinner()) {

alert(`Player ${this.currentPlayer} wins!`);

} else if (!this.board.includes(null)) {

alert('Draw!');

}

this.currentPlayer = this.currentPlayer === 'X' ? 'O' : 'X';

this.render();

}

checkWinner() {

const winningCombinations = [

[0, 1, 2], [3, 4, 5], [6, 7, 8], // rows

[0, 3, 6], [1, 4, 7], [2, 5, 8], // columns

[0, 4, 8], [2, 4, 6] // diagonals

];

for (const [a, b, c] of winningCombinations) {

if (this.board[a] && this.board[a] === this.board[b] && this.board[a] === this.board[c]) {

return true;

}

}

return false;

}

reset() {

this.board.fill(null);

this.currentPlayer = 'X';

this.render();

}

render() {

const boardElement = document.getElementById('board');

boardElement.innerHTML = '';

this.board.forEach((cell, index) => {

const cellElement = document.createElement('div');

cellElement.className = 'cell';

cellElement.textContent = cell;

cellElement.onclick = () => this.handleClick(index);

boardElement.appendChild(cellElement);

});

}

}

const game = new TicTacToe();

document.getElementById('reset-button').onclick = () => game.reset();

```

### 8. Fetch and Display API Data

\*\*Question:\*\*

Create a function that fetches data from a public API (e.g., GitHub Users API) and displays it on the web page. The function should handle loading states and errors gracefully.

\*\*Requirements:\*\*

- Fetch data from a public API

- Display the data

- Handle loading states

- Handle errors

\*\*Example:\*\*

```javascript

async function fetchAndDisplayUser(username) {

const container = document.getElementById('user-container');

container.innerHTML = 'Loading...';

try {

const response = await fetch(`https://api.github.com/users/${username}`); if (!response.ok) {

throw new Error('User not found');

}

const user = await response.json();

container.innerHTML = `

<h2>${user.name}</h2>

<img src="${user.avatar\_url}" alt="${user.name}" width="100">

<p>${user.bio}</p>

<p>Followers: ${user.followers}</p>

`;

} catch (error) {

container.innerHTML = `Error: ${error.message}`;

}

}

document.getElementById('fetch-user-button').onclick = () => {

const username = document.getElementById('username-input').value;

if (username) {

fetchAndDisplayUser(username);

}

};

```

### 9. Priority Queue with Custom Comparator

\*\*Question:\*\*

Implement a Priority Queue class in JavaScript that accepts a custom comparator function to determine the priority of elements.

\*\*Requirements:\*\*

- Enqueue elements with a priority

- Dequeue elements in order of priority

- Use a custom comparator for priority

\*\*Example:\*\*

```javascript

class PriorityQueue {

constructor(comparator = (a, b) => a - b) {

this.comparator = comparator;

this.items = [];

}

enqueue(item) {

this.items.push(item);

this.items.sort(this.comparator);

}

dequeue() {

return this.items.shift();

}

peek() {

return this.items[0];

}

isEmpty() {

return this.items.length === 0;

}

size() {

return this.items.length;

}

}

const pq = new PriorityQueue((a, b) => b.priority - a.priority);

pq.enqueue({ task: 'task1', priority: 2 });

pq.enqueue({ task: 'task2', priority: 1 });

pq.enqueue({ task: 'task3', priority: 3 });

console.log(pq.dequeue()); // { task: 'task3', priority: 3 }

console.log(pq.dequeue()); // { task: 'task1', priority: 2 }

console.log(pq.dequeue()); // { task: 'task2', priority: 1 }

```

### 10. Nested Checkboxes

\*\*Question:\*\*

Implement a component with nested checkboxes where checking a parent checkbox will automatically check all of its child checkboxes and vice versa. If any child checkbox is unchecked, the parent should also be unchecked.

\*\*Example:\*\*

```javascript

class CheckboxTree {

constructor(data) {

this.data = data;

this.render();

}

toggleCheckbox(node) {

node.checked = !node.checked;

if (node.children) {

node.children.forEach(child => this.toggleCheckboxRecursive(child, node.checked)); }

this.render();

}

toggleCheckboxRecursive(node, checked) {

node.checked = checked;

if (node.children) {

node.children.forEach(child => this.toggleCheckboxRecursive(child, checked)); }

}

render() {

const container = document.getElementById('checkbox-tree');

container.innerHTML = '';

const renderNode = (node, parentElement) => {

const checkbox = document.createElement('input');

checkbox.type = 'checkbox';

checkbox.checked = node.checked;

checkbox.onchange = () => this.toggleCheckbox(node);

const label = document.createElement('label');

label.appendChild(checkbox);

label.appendChild(document.createTextNode(node.label));

const div = document.createElement('div');

div.appendChild(label);

parentElement.appendChild(div);

if (node.children) {

node.children.forEach(child => renderNode(child, div));

}

};

this.data.forEach(node => renderNode(node, container));

}

}

const data = [

{

label: 'Parent 1',

checked: false,

children: [

{ label: 'Child 1.1', checked: false },

{ label: 'Child 1.2', checked: false },

]

},

{

label: 'Parent 2',

checked: false,

children: [

{ label: 'Child 2.1', checked: false },

{

label: 'Child 2.2', checked: false,

children: [

{ label: 'Grandchild 2.2.1', checked: false },

{ label: 'Grandchild 2.2.2', checked: false }

]

}

]

}

];

const tree = new CheckboxTree(data);

```

### 11. Paginated Widget

\*\*Question:\*\*

Implement a paginated widget that displays a list of addresses. The widget should show 5 addresses per page with the ability to navigate to the previous and next pages.

\*\*Example:\*\*

```javascript

class PaginatedWidget {

constructor(addresses) {

this.addresses = addresses;

this.currentPage = 1;

this.addressesPerPage = 5;

this.render();

}

changePage(page) {

if (page < 1 || page > this.totalPages()) return;

this.currentPage = page;

this.render();

}

totalPages() {

return Math.ceil(this.addresses.length / this.addressesPerPage);

}

render() {

const container = document.getElementById('address-container');

container.innerHTML = '';

const start = (this.currentPage - 1) \* this.addressesPerPage;

const end = start + this.addressesPerPage;

const addressesToShow = this.addresses.slice(start, end);

addressesToShow.forEach(address => {

const div = document.createElement('div');

div.textContent = address;

container.appendChild(div);

});

const pagination = document.createElement('div');

pagination.className = 'pagination';

const prevButton = document.createElement('button');

prevButton.textContent = 'Previous';

prevButton.onclick = () => this.changePage(this.currentPage - 1);

pagination.appendChild(prevButton);

const nextButton = document.createElement('button');

nextButton.textContent = 'Next';

nextButton.onclick = () => this.changePage(this.currentPage + 1);

pagination.appendChild(nextButton);

container.appendChild(pagination);

}

}

const addresses = [

'Address 1', 'Address 2', 'Address 3', 'Address 4', 'Address 5',

'Address 6', 'Address 7', 'Address 8', 'Address 9', 'Address 10',

'Address 11', 'Address 12', 'Address 13', 'Address 14', 'Address 15'

];

const widget = new PaginatedWidget(addresses);

```

### 12. Dynamic Table Generator

\*\*Question:\*\*

Create a form where users can input the number of rows and columns for a table, and upon submission, generate a table with cells filled in a specific pattern.

\*\*Example:\*\*

```javascript

document.getElementById('generate-table').onclick = () => {

const rows = parseInt(document.getElementById('rows').value);

const cols = parseInt(document.getElementById('cols').value);

if (isNaN(rows) || isNaN(cols)) {

alert('Please enter valid numbers for rows and columns.');

return;

}

const table = document.createElement('table');

let counter = 1;

for (let i = 0; i < rows; i++) {

const tr = document.createElement('tr');

for (let j = 0; j < cols; j++) {

const td = document.createElement('td');

td.textContent = counter++;

tr.appendChild(td);

}

table.appendChild(tr);

}

const tableContainer = document.getElementById('table-container');

tableContainer.innerHTML = '';

tableContainer.appendChild(table);

};

// HTML

// <input type="text" id="rows" placeholder="Number of rows">

// <input type="text" id="cols" placeholder="Number of columns">

// <button id="generate-table">Generate Table</button>

// <div id="table-container"></div>

```

### 13. Throttling Function

\*\*Question:\*\*

Implement a throttling function that limits the number of times a given function can be called within a specified time frame (e.g., once every 100 milliseconds).

\*\*Example:\*\*

```javascript

function throttle(fn, wait) {

let isThrottled = false, lastArgs, lastThis;

function wrapper() {

if (isThrottled) {

lastArgs = arguments;

lastThis = this;

return;

}

fn.apply(this, arguments);

isThrottled = true;

setTimeout(() => {

isThrottled = false;

if (lastArgs) {

wrapper.apply(lastThis, lastArgs);

lastArgs = lastThis = null;

}

}, wait);

}

return wrapper;

}

const log = throttle((message) => console.log(message), 1000);

document.getElementById('throttle-button').onclick = () => log('Button clicked!');

// HTML

// <button id="throttle-button">Click me</button>

```

### 14. Fetch and Display API Data with Loading and Error Handling

\*\*Question:\*\*

Create a function that fetches user data from the GitHub API and displays it on the page. The function should handle loading states and errors gracefully.

\*\*Example:\*\*

```javascript

async function fetchGitHubUser(username) {

const container = document.getElementById('user-container');

container.innerHTML = 'Loading...';

try {

const response = await fetch(`https://api.github.com/users/${username}`); if (!response.ok) throw new Error('User not found');

const user = await response.json();

container.innerHTML = `

<h2>${user.name}</h2>

<img src="${user.avatar\_url}" alt="${user.name}" width="100">

<p>${user.bio}</p>

<p>Followers: ${user.followers}</p>

`;

} catch (error) {

container.innerHTML = `Error: ${error.message}`;

}

}

document.getElementById('fetch-user-button').onclick = () => {

const username = document.getElementById('username-input').value;

fetchGitHubUser(username);

};

// HTML

// <input type="text" id="username-input" placeholder="GitHub username">

// <button id="fetch-user-button">Fetch User</button>

// <div id="user-container"></div>

```

These questions test various advanced JavaScript concepts, including closures, asynchronous programming, DOM manipulation, event handling, and dynamic content generation, providing a comprehensive assessment of a senior JavaScript developer's skills.

### 15. Debounce Function Implementation

\*\*Question:\*\*

Implement a debounce function that delays the execution of the input function until after a specified delay has passed since the last time the debounced function was invoked.

\*\*Example:\*\*

```javascript

function debounce(fn, delay) {

let timeoutID;

return function(...args) {

if (timeoutID) {

clearTimeout(timeoutID);

}

timeoutID = setTimeout(() => {

fn.apply(this, args);

}, delay);

};

}

const log = debounce((message) => console.log(message), 1000);

document.getElementById('debounce-button').onclick = () => log('Button clicked!');

// HTML

// <button id="debounce-button">Click me</button>

```

### 16. Dynamic Table with Sorting and Searching

\*\*Question:\*\*

Create a dynamic table that can be sorted by clicking on the column headers and includes a search bar to filter the table rows based on the input.

\*\*Example:\*\*

```javascript

class DynamicTable {

constructor(data) {

this.data = data;

this.filteredData = [...data];

this.sortOrder = 1; // 1 for ascending, -1 for descending

this.render();

}

sortTable(column) {

this.filteredData.sort((a, b) => {

if (a[column] < b[column]) return -1 \* this.sortOrder;

if (a[column] > b[column]) return 1 \* this.sortOrder;

return 0;

});

this.sortOrder \*= -1;

this.render();

}

searchTable(query) {

this.filteredData = this.data.filter(row =>

Object.values(row).some(val =>

val.toString().toLowerCase().includes(query.toLowerCase())

)

);

this.render();

}

render() {

const container = document.getElementById('table-container');

container.innerHTML = '';

const table = document.createElement('table');

const thead = document.createElement('thead');

const tr = document.createElement('tr');

Object.keys(this.data[0]).forEach(column => {

const th = document.createElement('th');

th.textContent = column;

th.onclick = () => this.sortTable(column);

tr.appendChild(th);

});

thead.appendChild(tr);

table.appendChild(thead);

const tbody = document.createElement('tbody');

this.filteredData.forEach(row => {

const tr = document.createElement('tr');

Object.values(row).forEach(value => {

const td = document.createElement('td');

td.textContent = value;

tr.appendChild(td);

});

tbody.appendChild(tr);

});

table.appendChild(tbody);

container.appendChild(table);

}

}

const data = [

{ Name: 'John', Age: 28, Country: 'USA' },

{ Name: 'Anna', Age: 22, Country: 'Sweden' },

{ Name: 'Mike', Age: 32, Country: 'Canada' },

];

const table = new DynamicTable(data);

document.getElementById('search-input').oninput = (e) =>

table.searchTable(e.target.value);

// HTML

// <input type="text" id="search-input" placeholder="Search">

// <div id="table-container"></div>

```

### 17. Fetch and Display Weather Data with Caching

\*\*Question:\*\*

Create a function that fetches weather data from a weather API and displays it on the page. Implement caching so that if the same city is requested within 10 minutes, the cached data is used instead of making a new API request.

\*\*Example:\*\*

```javascript

class WeatherApp {

constructor(apiKey) {

this.apiKey = apiKey;

this.cache = {};

}

async fetchWeather(city) {

const cacheKey = city.toLowerCase();

const cachedData = this.cache[cacheKey];

if (cachedData && (Date.now() - cachedData.timestamp < 10 \* 60 \* 1000)) { this.displayWeather(cachedData.data);

return;

}

const response = await fetch(`https://api.openweathermap.org/data/2.5/weather? q=${city}&appid=${this.apiKey}`);

if (!response.ok) {

this.displayError('City not found');

return;

}

const data = await response.json();

this.cache[cacheKey] = { data, timestamp: Date.now() };

this.displayWeather(data);

}

displayWeather(data) {

const container = document.getElementById('weather-container');

container.innerHTML = `

<h2>${data.name}</h2>

<p>Temperature: ${Math.round(data.main.temp - 273.15)}°C</p>

<p>Weather: ${data.weather[0].description}</p>

`;

}

displayError(message) {

const container = document.getElementById('weather-container');

container.innerHTML = `<p>Error: ${message}</p>`;

}

}

const app = new WeatherApp('YOUR\_API\_KEY');

document.getElementById('fetch-weather-button').onclick = () => {

const city = document.getElementById('city-input').value;

app.fetchWeather(city);

};

// HTML

// <input type="text" id="city-input" placeholder="Enter city">

// <button id="fetch-weather-button">Fetch Weather</button>

// <div id="weather-container"></div>

```

### 18. Create a Tic-Tac-Toe Game

\*\*Question:\*\*

Implement a simple Tic-Tac-Toe game using vanilla JavaScript, HTML, and CSS. The game should track the current player, check for win conditions, and allow players to reset the game.

\*\*Example:\*\*

```javascript

class TicTacToe {

constructor() {

this.board = Array(9).fill(null);

this.currentPlayer = 'X';

this.render();

}

makeMove(index) {

if (this.board[index] || this.checkWinner()) return;

this.board[index] = this.currentPlayer;

this.currentPlayer = this.currentPlayer === 'X' ? 'O' : 'X';

this.render();

}

checkWinner() {

const winningCombinations = [

[0, 1, 2], [3, 4, 5], [6, 7, 8],

[0, 3, 6], [1, 4, 7], [2, 5, 8],

[0, 4, 8], [2, 4, 6]

];

for (const [a, b, c] of winningCombinations) {

if (this.board[a] && this.board[a] === this.board[b] && this.board[a] === this.board[c]) {

return this.board[a];

}

}

return null;

}

resetGame() {

this.board.fill(null);

this.currentPlayer = 'X';

this.render();

}

render() {

const container = document.getElementById('tic-tac-toe');

container.innerHTML = '';

this.board.forEach((cell, index) => {

const div = document.createElement('div');

div.className = 'cell';

div.textContent = cell;

div.onclick = () => this.makeMove(index);

container.appendChild(div);

});

const winner = this.checkWinner();

if (winner) {

const message = document.createElement('div');

message.textContent = `Player ${winner} wins!`;

container.appendChild(message);

}

const resetButton = document.createElement('button');

resetButton.textContent = 'Reset';

resetButton.onclick = () => this.resetGame();

container.appendChild(resetButton);

}

}

new TicTacToe();

// HTML

// <div id="tic-tac-toe"></div>

// CSS

// .cell {

// width: 60px;

// height: 60px;

// display: inline-block;

// text-align: center;

// line-height: 60px;

// border: 1px solid #000;

// font-size: 24px;

// }

// #tic-tac-toe {

// display: grid;

// grid-template-columns: repeat(3, 60px);

// gap: 5px;

// }

```

### 19. Rate Limiter

\*\*Question:\*\*

Implement a rate limiter that caps the number of requests to an API endpoint to N requests per minute with a rolling window.

\*\*Example:\*\*

```javascript

class RateLimiter {

constructor(limit, interval) {

this.limit = limit;

this.interval = interval;

this.requests = [];

}

isAllowed() {

const now = Date.now();

this.requests = this.requests.filter(timestamp => now - timestamp < this.interval);

if (this.requests.length < this.limit) {

this.requests.push(now);

return true;

}

return false;

}

}

const limiter = new RateLimiter(5, 60000);

document.getElementById('rate-limit-button').onclick = () => {

if (limiter.isAllowed()) {

console.log('Request allowed');

// Simulate API request

} else {

console.log('Rate limit exceeded');

}

};

// HTML

// <button id="rate-limit-button">Make Request</button>

```

### 20. Virtualized List Rendering

\*\*Question:\*\*

Implement a virtualized list component that only renders visible items in a list to optimize performance for large datasets. Use vanilla JavaScript and implement scroll handling.

\*\*Example:\*\*

```javascript

class VirtualizedList {

constructor(containerId, items, itemHeight, visibleCount) {

this.container = document.getElementById(containerId);

this.items = items;

this.itemHeight = itemHeight;

this.visibleCount = visibleCount;

this.startIndex = 0;

this.endIndex = visibleCount;

this.render();

this.container.addEventListener('scroll', this.handleScroll.bind(this)); }

handleScroll() {

const scrollTop = this.container.scrollTop;

this.startIndex = Math.floor(scrollTop / this.itemHeight);

this.endIndex = this.startIndex + this.visibleCount;

this.render();

}

render() {

this.container.innerHTML = '';

const fragment = document.createDocumentFragment();

for (let i = this.startIndex; i < this.endIndex && i < this.items.length; i++) { const item = document.createElement('div');

item.className = 'item';

item.style.height = `${this.itemHeight}px`;

item.textContent = this.items[i];

fragment.appendChild(item);

}

this.container.appendChild(fragment);

}

}

const items = Array.from({ length: 10000 }, (\_, i) => `Item ${i + 1}`);

new VirtualizedList('virtual-list-container', items, 30, 20);

// HTML

// <div id="virtual-list-container" style="height: 600px; overflow-y: auto;"></div>

// CSS

// .item {

// box-sizing: border-box;

// padding: 8px;

// border-bottom: 1px solid #ccc;

// }

```

### 21. Custom Event Emitter

\*\*Question:\*\*

Implement a custom event emitter class in JavaScript that allows subscribing to events, emitting events, and removing event listeners.

\*\*Example:\*\*

```javascript

class EventEmitter {

constructor() {

this.events = {};

}

on(event, listener) {

if (!this.events[event]) {

this.events[event] = [];

}

this.events[event].push(listener);

}

off(event, listener) {

if (!this.events[event]) return;

this.events[event] = this.events[event].filter(l => l !== listener);

}

emit(event, ...args) {

if (!this.events[event]) return;

this.events[event].forEach(listener => listener(...args));

}

}

// Usage example

const emitter = new EventEmitter();

function onFoo(data) {

console.log('foo event:', data);

file:///C:/Users/prave/AppData/Local/Temp/ccdf6992-b7b4-442c-b80e-82beab0eacb5\_b493b1375914d0c0477d5f3d7512d7b3b8d12087190ab… 21/34

29/05/2024, 13:37 ChatGPT Data Export

}

emitter.on('foo', onFoo);

emitter.emit('foo', { some: 'data' }); // Output: foo event: { some: 'data' }

emitter.off('foo', onFoo);

emitter.emit('foo', { some: 'data' }); // No output

```

### 22. Dependency Injection

\*\*Question:\*\*

Create a simple dependency injection system in JavaScript. It should allow defining dependencies and resolving them at runtime.

\*\*Example:\*\*

```javascript

class Container {

constructor() {

this.services = new Map();

}

register(name, definition, dependencies) {

this.services.set(name, { definition, dependencies });

}

resolve(name) {

const target = this.services.get(name);

if (!target) {

throw new Error(`Service not found: ${name}`);

}

if (!target.instance) {

const { definition, dependencies } = target;

const resolvedDependencies = dependencies.map(dep => this.resolve(dep)); target.instance = new definition(...resolvedDependencies);

}

return target.instance;

}

}

class Logger {

log(message) {

console.log(message);

}

}

class UserService {

constructor(logger) {

this.logger = logger;

}

getUser() {

this.logger.log('User fetched');

return { name: 'John Doe' };

}

}

const container = new Container();

container.register('logger', Logger, []);

container.register('userService', UserService, ['logger']);

const userService = container.resolve('userService');

console.log(userService.getUser()); // Output: User fetched { name: 'John Doe' } ```

### 23. Dynamic Form Builder

\*\*Question:\*\*

Create a dynamic form builder that generates a form based on a JSON schema. The form should support validation and handle form submissions.

\*\*Example:\*\*

```javascript

class FormBuilder {

constructor(schema, containerId) {

this.schema = schema;

this.container = document.getElementById(containerId);

this.render();

}

render() {

this.container.innerHTML = '';

const form = document.createElement('form');

this.schema.fields.forEach(field => {

const fieldElement = this.createField(field);

form.appendChild(fieldElement);

});

const submitButton = document.createElement('button');

submitButton.type = 'submit';

submitButton.textContent = 'Submit';

form.appendChild(submitButton);

form.onsubmit = this.handleSubmit.bind(this);

this.container.appendChild(form);

}

createField(field) {

const fieldWrapper = document.createElement('div');

const label = document.createElement('label');

label.textContent = field.label;

fieldWrapper.appendChild(label);

const input = document.createElement('input');

input.type = field.type;

input.name = field.name;

input.required = field.required;

fieldWrapper.appendChild(input);

return fieldWrapper;

}

handleSubmit(event) {

event.preventDefault();

const formData = new FormData(event.target);

const data = {};

formData.forEach((value, key) => {

data[key] = value;

});

if (this.validate(data)) {

console.log('Form data:', data);

} else {

console.log('Validation failed');

}

}

validate(data) {

for (const field of this.schema.fields) {

if (field.required && !data[field.name]) {

return false;

}

}

return true;

}

}

const schema = {

fields: [

{ label: 'Name', name: 'name', type: 'text', required: true },

{ label: 'Email', name: 'email', type: 'email', required: true },

{ label: 'Password', name: 'password', type: 'password', required: true } ]

};

new FormBuilder(schema, 'form-container');

// HTML

// <div id="form-container"></div>

```

### 24. Image Carousel with Lazy Loading

\*\*Question:\*\*

Implement an image carousel component with lazy loading for images. The carousel should allow users to navigate through images and load images only when they come into view.

\*\*Example:\*\*

```javascript

class ImageCarousel {

constructor(containerId, images) {

this.container = document.getElementById(containerId);

this.images = images;

this.currentIndex = 0;

this.render();

}

render() {

this.container.innerHTML = '';

const img = document.createElement('img');

img.className = 'carousel-image';

img.src = this.images[this.currentIndex];

img.loading = 'lazy';

this.container.appendChild(img);

const prevButton = document.createElement('button');

prevButton.textContent = 'Previous';

prevButton.onclick = this.prevImage.bind(this);

this.container.appendChild(prevButton);

const nextButton = document.createElement('button');

nextButton.textContent = 'Next';

nextButton.onclick = this.nextImage.bind(this);

this.container.appendChild(nextButton);

}

prevImage() {

this.currentIndex = (this.currentIndex - 1 + this.images.length) %

this.images.length;

this.render();

}

nextImage() {

this.currentIndex = (this.currentIndex + 1) % this.images.length;

this.render();

}

}

const images = [

'https://via.placeholder.com/400x300?text=Image+1',

'https://via.placeholder.com/400x300?text=Image+2',

'https://via.placeholder.com/400x300?text=Image+3'

];

new ImageCarousel('carousel-container', images);

// HTML

// <div id="carousel-container"></div>

// CSS

// .carousel-image {

// width: 400px;

// height: 300px;

// display: block;

// margin-bottom: 10px;

// }

```

### 25. Middleware Pipeline

\*\*Question:\*\*

Create a middleware pipeline in JavaScript that allows chaining of multiple middleware functions to process a request and response object. The middleware should be executed in order, passing control to the next middleware in the chain.

\*\*Example:\*\*

```javascript

class MiddlewarePipeline {

constructor() {

this.middlewares = [];

}

use(middleware) {

this.middlewares.push(middleware);

}

execute(context) {

const compose = (middlewares) => {

return function (context, next) {

let index = -1;

function dispatch(i) {

if (i <= index) return Promise.reject(new Error('next() called multiple times'));

index = i;

let fn = middlewares[i];

if (i === middlewares.length) fn = next

**ChatGPT**

;

if (!fn) return Promise.resolve();

try {

return Promise.resolve(fn(context, () => dispatch(i + 1)));

} catch (err) {

return Promise.reject(err);

}

}

return dispatch(0);

};

};

return compose(this.middlewares)(context, () => Promise.resolve());

}

}

// Usage example

const pipeline = new MiddlewarePipeline();

pipeline.use(async (ctx, next) => {

console.log('Middleware 1 start');

ctx.data.push(1);

await next();

console.log('Middleware 1 end');

});

pipeline.use(async (ctx, next) => {

console.log('Middleware 2 start');

ctx.data.push(2);

await next();

console.log('Middleware 2 end');

});

pipeline.use(async (ctx, next) => {

console.log('Middleware 3 start');

ctx.data.push(3);

await next();

console.log('Middleware 3 end');

});

const context = { data: [] };

pipeline.execute(context).then(() => {

console.log('Pipeline completed', context);

}).catch(err => {

console.error('Pipeline error', err);

});

```

These questions aim to test a tech lead's ability to manage complex interactions, optimize performance, design scalable solutions, and demonstrate in-depth knowledge of JavaScript.

### 26. Rate Limiter with Redis

\*\*Question:\*\*

Implement a rate limiter using Redis in JavaScript. The rate limiter should allow a maximum of 100 requests per minute per user. Use the `ioredis` library for interacting with Redis.

\*\*Example:\*\*

```javascript

const Redis = require('ioredis');

const redis = new Redis();

class RateLimiter {

constructor(maxRequests, windowMs) {

this.maxRequests = maxRequests;

this.windowMs = windowMs;

}

async isAllowed(userId) {

const key = `rate\_limiter:${userId}`;

const current = await redis.incr(key);

if (current === 1) {

await redis.pexpire(key, this.windowMs);

}

return current <= this.maxRequests;

}

}

// Usage example

const rateLimiter = new RateLimiter(100, 60000);

async function handleRequest(userId) {

const allowed = await rateLimiter.isAllowed(userId);

if (allowed) {

console.log('Request allowed');

} else {

console.log('Request rate limited');

}

}

handleRequest('user123');

```

### 27. Dynamic Module Loader

\*\*Question:\*\*

Implement a dynamic module loader in JavaScript that can load JavaScript modules dynamically based on user input. Use the `import` function to achieve this.

\*\*Example:\*\*

```javascript

class ModuleLoader {

constructor(basePath) {

this.basePath = basePath;

}

async loadModule(moduleName) {

try {

const module = await import(`${this.basePath}/${moduleName}.js`);

return module;

} catch (error) {

console.error(`Failed to load module: ${moduleName}`, error);

throw error;

}

}

}

// Usage example

const loader = new ModuleLoader('/modules');

loader.loadModule('exampleModule')

.then(module => {

module.default();

})

.catch(error => {

console.error('Error loading module:', error);

});

```

### 28. Transactional State Management

\*\*Question:\*\*

Implement a transactional state management system in JavaScript that allows changes to state to be committed or rolled back. Use a stack to manage the transactions.

\*\*Example:\*\*

```javascript

class TransactionalState {

constructor(initialState = {}) {

this.state = { ...initialState };

this.transactions = [];

}

begin() {

this.transactions.push({ ...this.state });

}

commit() {

if (this.transactions.length === 0) {

throw new Error('No transaction to commit');

}

this.transactions.pop();

}

rollback() {

if (this.transactions.length === 0) {

throw new Error('No transaction to rollback');

}

this.state = this.transactions.pop();

}

set(key, value) {

if (this.transactions.length > 0) {

this.state[key] = value;

} else {

throw new Error('No active transaction');

}

}

get(key) {

return this.state[key];

}

}

// Usage example

const state = new TransactionalState({ a: 1, b: 2 });

state.begin();

state.set('a', 10);

console.log(state.get('a')); // 10

state.rollback();

console.log(state.get('a')); // 1

state.begin();

state.set('a', 20);

state.commit();

console.log(state.get('a')); // 20

```

### 29. WebSocket Server with Authentication

\*\*Question:\*\*

Implement a WebSocket server in Node.js with authentication. Use the `ws` library for WebSocket communication. The server should authenticate clients using a token passed during the connection.

\*\*Example:\*\*

```javascript

const WebSocket = require('ws');

const jwt = require('jsonwebtoken');

const wss = new WebSocket.Server({ port: 8080 });

function authenticate(token) {

try {

return jwt.verify(token, 'your\_secret\_key');

} catch (err) {

return null;

}

}

wss.on('connection', (ws, req) => {

const token = req.url.split('token=')[1];

const user = authenticate(token);

if (!user) {

ws.close();

return;

}

ws.on('message', (message) => {

console.log('received:', message);

ws.send(`Hello, ${user.name}`);

});

ws.send('Welcome to the WebSocket server!');

});

// Usage example

const ws = new WebSocket('ws://localhost:8080?token=your\_jwt\_token');

ws.on('open', () => {

ws.send('Hello Server!');

});

ws.on('message', (data) => {

console.log(data);

});

```

### 30. Priority Queue

\*\*Question:\*\*

Implement a priority queue in JavaScript. The priority queue should allow enqueueing items with a priority and dequeuing items in order of their priority (higher priority items are dequeued first).

\*\*Example:\*\*

```javascript

class PriorityQueue {

constructor() {

this.queue = [];

}

enqueue(item, priority) {

this.queue.push({ item, priority });

this.queue.sort((a, b) => b.priority - a.priority);

}

dequeue() {

return this.queue.shift().item;

}

peek() {

return this.queue[0]?.item;

}

isEmpty() {

return this.queue.length === 0;

}

}

// Usage example

const pq = new PriorityQueue();

pq.enqueue('low priority task', 1);

pq.enqueue('high priority task', 10);

pq.enqueue('medium priority task', 5);

console.log(pq.dequeue()); // high priority task

console.log(pq.dequeue()); // medium priority task

console.log(pq.dequeue()); // low priority task

```

### 31. GraphQL Server with Node.js

\*\*Question:\*\*

Implement a simple GraphQL server in Node.js that allows querying and mutating user data. Use the `apollo-server` library for the GraphQL server implementation.

\*\*Example:\*\*

```javascript

const { ApolloServer, gql } = require('apollo-server');

// Sample data

const users = [

{ id: 1, name: 'Alice' },

{ id: 2, name: 'Bob' },

];

// GraphQL schema

const typeDefs = gql`

type User {

id: ID!

name: String!

}

type Query {

users: [User]

user(id: ID!): User

}

type Mutation {

addUser(name: String!): User

updateUser(id: ID!, name: String!): User

}

`;

// GraphQL resolvers

const resolvers = {

Query: {

users: () => users,

user: (\_, { id }) => users.find(user => user.id === parseInt(id)),

},

Mutation: {

addUser: (\_, { name }) => {

const newUser = { id: users.length + 1, name };

users.push(newUser);

return newUser;

},

updateUser: (\_, { id, name }) => {

const user = users.find(user => user.id === parseInt(id));

if (user) {

user.name = name;

return user;

}

return null;

},

},

};

// Creating the GraphQL server

const server = new ApolloServer({ typeDefs, resolvers });

server.listen().then(({ url }) => {

console.log(`Server ready at ${url}`);

});

// Usage example

// Query: { users { id, name } }

// Mutation: mutation { addUser(name: "Charlie") { id, name } }

```

### 32. Recursive Descent Parser

\*\*Question:\*\*

Implement a recursive descent parser for a simple arithmetic expression evaluator. The parser should handle addition, subtraction, multiplication, and division.

\*\*Example:\*\*

```javascript

class Parser {

constructor(expression) {

this.tokens = expression.match(/\d+|[+\-\*/()]/g);

this.current = 0;

}

parse() {

return this.expression();

}

expression() {

let node = this.term();

while (this.match('+', '-')) {

const operator = this.previous();

const right = this.term();

node = { type: 'BinaryExpression', operator, left: node, right };

}

return node;

}

term() {

let node = this.factor();

while (this.match('\*', '/')) {

const operator = this.previous();

const right = this.factor();

node = { type: 'BinaryExpression', operator, left: node, right };

}

return node;

}

factor() {

if (this.match('(')) {

const expr = this.expression();

this.consume(')');

return expr;

}

if (this.match(/\d+/)) {

return { type: 'Literal', value: Number(this.previous()) };

}

}

match(...expected) {

if (this.check(...expected)) {

this.advance();

return true;

}

return false;

}

check(...expected) {

if (this.isAtEnd()) return false;

return expected.some(exp => exp === this.peek() || (exp instanceof RegExp && exp.test(this.peek())));

}

advance() {

if (!this.isAtEnd()) this.current++;

}

consume(expected) {

if (this.check(expected)) {

this.advance();

return;

}

throw new Error(`Expected ${expected} but found ${this.peek()}`);

}

previous() {

return

**ChatGPT**

this.tokens[this.current - 1];

}

peek() {

return this.tokens[this.current];

}

isAtEnd() {

return this.current >= this.tokens.length;

}

}

// Usage example

const parser = new Parser('3 + 5 \* (10 - 2)');

const ast = parser.parse();

console.log(JSON.stringify(ast, null, 2));

```

These questions are designed to test a tech lead's ability to handle advanced JavaScript concepts, design scalable solutions, and manage complex state and interactions.