

Up-to-date Questions and Answers from authentic resources to improve knowledge and pass the exam at very first attempt. ----- Guaranteed.

KILL EXAMS

Javascript-Developer-I MCQs
Javascript-Developer-I Exam Questions
Javascript-Developer-I Practice Test
Javascript-Developer-I TestPrep
Javascript-Developer-I Study Guide



killexams.com

SalesForce



Javascript-Developer-I

Salesforce Certified JavaScript Developer I

ORDER FULL VERSION

https://killexams.com/pass4sure/exam-detail/Javascript-Developer-I

Question: 1499

Salesforce custom validation LWC: const inputVal = 'false'; const isFlag = inputVal == true; Given string 'false' comparison, what is isFlag due to coercion?

A. false, 'false' to boolean false != true
B. true, string to true always
C. false, loose == coerces 'false' to false
D. true, 'false' truthy string

Answer: C
Explanation: == true coerces left to boolean: ToBoolean('false') false (string 'false' falsy), false == true false. ToPrimitive then boolean. isFlag false.

Question: 1500

A Node.js server needs to read a 10GB file, calculate its SHA-256 hash, and then compress the file using Gzip. The server has only 512MB of RAM available. Which implementation strategy is correct? (Select two)

A. fs.createReadStream piped into crypto.createHash and zlib.createGzip
B. fs.promises.readFile followed by crypto.createHash and zlib.gzip
C. Using pipeline() from the stream module to connect the read stream, hash stream, and gzip stream
D. Creating a Buffer of the entire file, then applying synchronous crypto and zlib functions
E. Reading the file in chunks of 1GB using fs.readSync and storing them in an array

Answer: A,C
Explanation: Streams are the only way to process data larger than the available RAM. pipeline() is preferred over .pipe() because it handles cleanup and provides a callback for error handling at the end of the entire chain. crypto.createHash and zlib.createGzip both return transform streams that can process data incrementally.

Question: 1501

A developer is working with a JSON response where the field names contain spaces.
const resp = '{ "Account Name": "Universal Containers" }';
const data = JSON.parse(resp);
How can the developer access the Account Name? (Select two)

A. data["Account Name"]
B. data.get("Account Name")
C. const { "Account Name": name } = data;
D. data.Account Name
E. data.'Account Name'

Answer: A,C
Explanation: When property names contain spaces or special characters, bracket notation is required for access. Alternatively, destructuring with a string literal key and a new variable name allows for extraction.

Question: 1502

A Salesforce integration middleware uses a factory function to create API client instances for different external systems. Each client must have its own configuration, but also share a common set of utility methods (like logRequest and handleError). The factory should minimize memory usage by having the shared methods exist only once in memory, rather than being copied to each instance. Which pattern should the factory employ?

A. function createApiClient(config) { const client = { config, logRequest() { /* ... */ }, handleError() { /* ... */ } }; return client; }
B. function createApiClient(config) { const client = Object.create(apiClientPrototype); client.config = config; return client; } const apiClientPrototype = { logRequest() { /* ... */ }, handleError() { /* ... */ } };
C. function createApiClient(config) { class ApiClient { constructor(config) { this.config = config; } logRequest() { /* ... */ } handleError() { /* ... */ } } return new ApiClient(config); }
D. function createApiClient(config) { return { config, __proto__: sharedMethods, logRequest: sharedMethods.logRequest, handleError: sharedMethods.handleError }; } const sharedMethods = { logRequest() { /* ... */ }, handleError() { /* ... */ } };

Answer: B
Explanation: Option uses explicit prototypal inheritance with Object.create. This pattern creates a new object that has apiClientPrototype as its prototype. The shared methods are defined on this prototype object, so they exist in memory only once and are delegated to via the prototype chain. Each instance gets its own config property. This is the most memory-efficient and classic JavaScript pattern for creating multiple objects that share behavior without duplicating function definitions.

Question: 1503

In a scenario where a Salesforce developer must query platform metadata like CPU load, memory RSS/heap, and uptime for auto-scaling LWC components based on org limits, then expose via a custom REST endpoint for External Services, which Node.js core module provides the necessary system metrics?

A. cluster module for worker stats
B. os module for system and process metrics
C. perf_hooks module for performance timing
D. process module for env variables only

Answer: B
Explanation: The os module delivers comprehensive metrics like os.freemem(), os.cpus(), and os.loadavg() essential for monitoring Salesforce org performance and triggering auto-scaling.

Question: 1504

Given the following code, what is the value of obj.count?
javascript
const obj = { count: 0 };
function increment() {
this.count++;
}
increment.bind(obj)();

A. 0
B. 1
C. NaN
D. undefined

Answer: B
Explanation: The .bind() method creates a new function that, when called, has its 'this' keyword set to the provided value (obj), correctly incrementing the property.

Question: 1505

Which initializations correctly use Symbols as non-enumerable, collision-free keys for internal state in an LWC class? (Select two)

A. const privateKey = Symbol("internal")
B. let state = { [Symbol.for("cache")]: new Map() }
C. const key = Symbol(); this[key] = value
D. let obj = Object.defineProperty({}, Symbol("hidden"), { value: 42 })
E. const sym = Symbol("same"); obj[sym] = obj[sym] ?? 0

Answer: A,C
Explanation: Unique Symbols created with Symbol() guarantee no key collisions. They are non-enumerable by default and ideal for private-like properties.

Question: 1506

In crypto-heavy workloads, which patterns optimize performance for repeated hashing? (Select three)

A. Reuse crypto.createHash('sha256') instance by calling .update() multiple times and .digest() once
B. Use crypto.createHash().update(data).digest() for each operation to avoid state issues
C. Prefer scrypt or argon2id via external libs for password hashing over pbkdf2
D. Use timingSafeEqual for comparing HMAC or hash digests to prevent timing attacks
E. Cache digest results in a Map keyed by input Buffer for identical inputs

Answer: A,D,E
Explanation: Reusing Hash instances reduces allocation overhead for incremental updates. timingSafeEqual prevents side-channel leaks. Caching identical inputs avoids redundant crypto operations in hot paths.

Question: 1507

WeakRef for cleanup:
javascript
class Watcher {
constructor(obj) {
this.ref = new WeakRef(obj);
}
check() { return this.ref.deref(); }
}
Usage? (Select two)

A. deref returns undefined after GC
B. Allows weak reference without leak
C. Strong reference if held elsewhere
D. Works with primitives

Answer: A,B
Explanation: WeakRef enables observing GC without preventing it.

Question: 1508

For an LWC infinite scroll loading Salesforce Leads in batches, you append new leads to existing ones immutably, then sort by CreatedDate descending only if the batch includes future dates (detected via some()). Which code avoids unnecessary sorts?

A. leads.some(l => new Date(l.CreatedDate) > new Date()) ? [...leads, ...newLeads].sort((a,b) => new Date(b.CreatedDate) - new Date(a.CreatedDate)) : [...leads, ...newLeads]
B. [...leads, ...newLeads.sort((a,B. => new Date(b.CreatedDate) - new Date(a.CreatedDate))]
C. const allLeads = [...leads, ...newLeads]; allLeads.some(l => new Date(l.CreatedDate) > new Date()) && allLeads.sort((a,b) => new Date(b.CreatedDate) - new Date(a.CreatedDate))
D. leads.concat(newLeads).sort((a,b) => some(l => l.CreatedDate > Date.now()) ? new Date(b.CreatedDate) - new Date(a.CreatedDate) : 0)

Answer: A
Explanation: some() checks for out-of-order dates before concatenating and sorting a new array, skipping the expensive sort when unnecessary. This conditional immutability optimizes performance in scroll-heavy LWCs with chronological Salesforce data.

Question: 1509

A developer is using String.prototype.replace() with a regular expression and a replacement function. What parameters are passed to the replacement function? (Select three)

A. The full match found by the regex.
B. Any captured groups from the regex.
C. The offset (index) where the match was found.
D. The entire original string.
E. The length of the match.

Answer: A,B,C
Explanation: The replacement function receives the match, followed by any capture groups, the numerical offset of the match within the string, and finally the string itself.

Question: 1510

A developer enhances Commerce Cloud product carousel with keyboard navigation using keydown events. Business logic requires wrapping focus from last to first item on ArrowRight at end. Tabindex management conflicts with Lightning focus trap. Which approach maintains accessibility while enabling carousel wrapping?

A. Custom roving tabindex with keydown ArrowRight modulus calculation
B. CSS scroll-snap with keyboard event.scrollIntoView() delegation

C. ARIA live region announcements during focus wrapping
D. IntersectionObserver for visible item focus management

Answer: A
Explanation: Implementing roving tabindex pattern sets tabindex=0 on current carousel item and -1 on others, with keydown handler calculating next index via (currentIndex + 1) % items.length, ensuring keyboard navigation wraps correctly while preserving Lightning focus management compatibility.

Question: 1511

JavaScriptlet x = [10, NaN, "", 0n, false, undefined, {valueOf: () => 0}];
let result = x.filter(Boolean);
Which values remain in result after evaluation? (Select three)

A. 10
B. NaN
C. {valueOf: () => 0}
D. 0n
E. false

Answer: A,B,C
Explanation: The Boolean function coerces each element to boolean. Numeric 10 is truthy. NaN is falsy but the coercion happens on the primitive value before the object wrapper matters. Any non-primitive object (even one whose valueOf returns falsy) is truthy. Empty string, 0n, false and undefined are falsy and filtered out.

Question: 1512

A developer is building a dashboard with dynamically loaded widgets. Each widget contains nested interactive elements (buttons, links, inputs). To meet performance requirements while handling clicks reliably across all current and future widgets, which approaches should be used? (Select All that Apply)

A. Add individual click listeners to every button and link after insertion
B. Attach a single click listener to the dashboard container with event delegation
C. Call event.preventDefault() in delegated handlers for anchor tags
D. Use event.stopImmediatePropagation() to block third-party widget scripts
E. Query the DOM with closest() to identify the specific widget target

Answer: B,C,E
Explanation: A single delegated listener on the container improves performance for dynamic content. preventDefault() is needed for anchors to avoid navigation. closest() helps traverse upward from

event.target to find the relevant widget context without fragile selectors.

Question: 1513

A developer observes that a UI is freezing during a heavy calculation. How can asynchronous programming help? (Select two)

A. By breaking the calculation into chunks and using setTimeout to schedule them
B. By moving the calculation to a Web Worker
C. By wrapping the synchronous calculation in an async function
D. By using Promise.resolve() around the calculation
E. By increasing the CPU priority of the JavaScript thread

Answer: A,B
Explanation: Simply wrapping code in an async function or Promise.resolve() does not make it asynchronous if the logic itself is synchronous; it will still block the main thread. Breaking the work into smaller pieces via setTimeout allows the event loop to handle UI updates between chunks. Web Workers run on a separate thread entirely.

Question: 1514

A developer is building a Lightning Web Component that processes user input from a form. The component receives a value that could be an empty string, null, or a number. The code uses the following condition: if (inputValue == 0). When the user submits an empty string, what will be the evaluation result?

A. The condition evaluates to false because empty strings are truthy
B. The condition evaluates to false because type coercion converts empty string to NaN
C. The condition evaluates to true because empty string coerces to 0 in numeric context
D. The condition throws a TypeError due to incompatible type comparison

Answer: C
Explanation: When using the loose equality operator (==), JavaScript performs type coercion. An empty string is coerced to the number 0 when compared with a numeric value. The abstract equality comparison algorithm converts the empty string to a number using ToNumber, which results in 0, making the comparison 0 == 0 evaluate to true.

Question: 1515

A developer wants to compress JSON responses in a Node.js API to reduce latency. Which core Node.js module should be used to implement Gzip or Deflate? (Select two)

A. crypto
B. http2
C. stream
D. url
E. zlib

Answer: C,E
Explanation: The zlib module provides compression functionality. Since it is implemented using streams, the stream module logic is essential for piping data through a compression filter.

Question: 1516

A developer is using a WeakMap to store private data related to DOM nodes. What are the key differences between Map and WeakMap? (Select two)

A. WeakMap keys must be objects
B. WeakMap is not enumerable (cannot be looped over)
C. WeakMap prevents its keys from being garbage collected
D. WeakMap has a size property
E. WeakMap supports primitive strings as keys

Answer: A,B
Explanation: WeakMap requires keys to be objects so they can be weakly held, allowing for garbage collection if no other references exist. Because of this weak reference, the contents of a WeakMap are unpredictable, so the API does not allow iteration or a size property.

Question: 1517

A developer needs to track customer order timestamps across different time zones in a Lightning Web Component. The component receives UTC timestamps from the server and must display them in the user's local timezone while maintaining the ability to calculate time differences in milliseconds. Which variable initialization approach correctly handles this requirement?

A. const orderDate = new Date(utcTimestamp); const displayTime = orderDate.toLocaleString(); const timeDiff = Date.now() - orderDate;
B. let orderDate = Date.parse(utcTimestamp); let displayTime = new Date(orderDate).toString(); let timeDiff = new Date() - orderDate;
C. let orderDate = new Date(utcTimestamp); let displayTime = orderDate.toLocaleDateString(); let timeDiff = orderDate.getTime() - Date.now();
D. var orderDate = new Date(utcTimestamp); var displayTime = orderDate.toISOString(); var timeDiff =

orderDate - new Date();

Answer: A
Explanation: Creating a Date object from a UTC timestamp automatically handles timezone conversion when using toLocaleString(), which formats the date according to the user's browser locale and timezone settings. The Date constructor parses the UTC timestamp correctly, and subtracting the Date object from Date.now() returns the difference in milliseconds because Date objects are coerced to their primitive number value (milliseconds since epoch) in arithmetic operations. Using const is appropriate here as the Date object reference doesn't need reassignment, and toLocaleString() provides both date and time in the user's local format.

Question: 1518

A Salesforce developer managing a monorepo with 200+ LWC bundles, native plugins, and shared utilities needs a package manager that supports workspace federation, lockfile optimization for CI/CD, and selective hoisting for Salesforce DX scratch orgs. Which Node.js Package Management solution fits?

A. Bower for frontend assets
B. npm with workspaces
C. pnpm with strict store
D. Yarn Berry plug'n'play

Answer: C
Explanation: pnpm's content-addressable store and workspace protocol enable efficient monorepo management with hard links, reducing scratch org deploy times by 70%+ for large Salesforce projects.

KILLEXAMS.COM

Killexams.com is a leading online platform specializing in high-quality certification exam preparation. Offering a robust suite of tools, including MCQs, practice tests, and advanced test engines, Killexams.com empowers candidates to excel in their certification exams. Discover the key features that make Killexams.com the go-to choice for exam success.



Exam Questions Based on Current Exam Objectives

Killexams.com provides exam questions aligned with the latest official exam objectives and latest syllabus. Our content is reviewed and updated regularly to reflect recent changes announced by certification vendors. By studying these questions, candidates will become cover the structure, difficulty level, and topic coverage of the actual exam, helping them prepare more effectively and efficiently.

Comprehensive Exam MCQs (PDF Format)

Killexams.com offers multiple-choice questions (MCQs) in easy-to-read PDF format, covering all major domains of the exam. Each PDF contains a structured collection of questions and verified answers designed to support focused study. These MCQs help candidates reinforce key concepts, identify knowledge gaps, and improve exam readiness through consistent practice.

Realistic Practice Tests (Online & Desktop)

To support hands-on preparation, Killexams.com provides practice tests through both an Online Test Engine and a Desktop Exam Simulator. These tools are designed to simulate a real exam environment, allowing candidates to practice under exam-like conditions. Performance tracking, test history, and result analysis help users evaluate their progress and focus on areas that need improvement.

Risk-Free Purchase Policy

Killexams.com follows a transparent and customer-friendly purchase policy. If users are not satisfied with the study materials, they may request assistance or a refund in accordance with our published terms and conditions. This policy reflects our commitment to customer satisfaction, fairness, and confidence in our preparation resources.

Regularly Updated Content

Our question bank is reviewed and updated on an ongoing basis to stay aligned with the latest exam outlines and vendor updates. This ensures candidates are studying relevant material and preparing with content that reflects current exam expectations, helping them stay confident and well-prepared.