LocalStorage

---

1. Technical Deep Dive: Must-Knows

Key Characteristics

- **Capacity:** Roughly **5MB – 10MB** per origin (protocol + domain + port). Exceeding this throws a QuotaExceededError.
- **Synchronous API:** This is critical. localStorage is **main-thread blocking**. If you read or write a massive JSON string, the UI will freeze until the operation completes.
- **Persistence:** Data has no expiration date. It persists until the user clears their cache or the app programmatically calls localStorage.clear().
- **Data Type:** Only stores **UTF-16 strings**. You must use JSON.stringify() and JSON.parse() for objects, which adds CPU overhead.
- **Scope:** Shared across all tabs/windows of the **same origin**.

The "Storage" Event

One often-overlooked feature is the storage event. It allows tabs to communicate.

```
window.addEventListener('storage', (event) => {
  if (event.key === 'theme') {
    applyTheme(event.newValue); // Syncs theme across all open tabs
  }
});
```

---

2. Security: The XSS Vulnerability

In a System Design interview, security is a top priority.

- **XSS (Cross-Site Scripting):** Because localStorage is accessible via any JavaScript on the page, if an attacker successfully injects a script, they can steal everything: JSON.stringify(localStorage).
- **The Recommendation: Never store sensitive data** (JWTs, session tokens, PII) in localStorage. Use **HttpOnly Cookies** for auth tokens to prevent JS access.

---

3. High-Frequency Interview Q&A

Q1: "When would you use localStorage vs. IndexedDB?"

Answer:

- **Use localStorage** for small, simple, non-sensitive data that requires no complex querying (e.g., user theme preference, a "sidebar-collapsed" state, or small UI caches).
- **Use IndexedDB** for large datasets (GBs), binary data (Blobs), or when you need **asynchronous** operations to avoid jank. If you're building an offline-first Trello or Google Docs, IndexedDB is the choice.

Q2: "How do you handle the 5MB limit in a production app?"

Answer:

1. **Try-Catch:** Always wrap setItem in a try-catch block to handle QuotaExceededError.
2. **LRU Cache:** Implement a "Least Recently Used" eviction policy. When storage is full, delete the oldest items.
3. **Data Compression:** Use libraries like lz-string to compress strings before saving.
4. **Fallback:** If localStorage is full, degrade gracefully by using in-memory storage (which clears on refresh).

Q3: "Is localStorage thread-safe across multiple tabs?"

Answer:
Technically, the browser manages the lock on the underlying file, so you won't "corrupt" the database file itself. However, you can encounter Race Conditions.

- *Example:* Two tabs read count: 5, both increment it to 6, and both write 6. The final value is 6 instead of 7.
- *Solution:* Use the storage event or a BroadCast Channel API to coordinate writes.

Q4: "How does localStorage impact performance?"

Answer:
It impacts the Critical Rendering Path. If your app blocks rendering to read a large configuration from localStorage during startup, your Time to Interactive (TTI) will suffer.

- **Optimization:** Read from localStorage once during initialization and keep it in an in-memory variable (like a React State or Redux store) for subsequent access.

Comparison Table for the Interviewer

Feature	LocalStorage	SessionStorage	Cookies
Size	5-10MB	5MB	4KB
Expiration	Never	Tab Close	Manually set
Access	Client Only	Client Only	Client & Server
Performance	Synchronous (Slow)	Synchronous (Slow)	Sent with every request (Network overhead)

Eviction Strategy:

The browser has no native eviction policy. Unlike the browser's **HTTP Cache** or **Memory Cache**—where the browser automatically deletes old files to make room for new ones—localStorage is strictly managed by the application. If you fill up that 5MB, it stays full until you manually clear it or the user wipes their browser data.

---

1. Implementing Custom Eviction (LRU)

If you are caching API responses in localStorage to improve load times, you must treat it like a mini-database and implement an **LRU (Least Recently Used)** cache.

How to design it:

- **Metadata:** Store objects with a lastAccessed timestamp or a version number.
- **The Process:** * Before writing new data, check the current size or catch the QuotaExceededError.
    - If full, sort all stored items by lastAccessed.
    - Delete the oldest 20% of entries.
    - Retry the write operation.

---

2. Expiration (TTL - Time to Live)

localStorage data lives forever, which is dangerous because you might serve "stale" (outdated) data to a user weeks later.

The Pattern: Wrap your data in an envelope that includes an expiry timestamp.

```
const setWithExpiry = (key, value, ttl) => {
  const item = {
    value: value,
    expiry: Date.now() + ttl, // ttl in milliseconds
  };
  localStorage.setItem(key, JSON.stringify(item));
};

const getWithExpiry = (key) => {
  const itemStr = localStorage.getItem(key);
  if (!itemStr) return null;

  const item = JSON.parse(itemStr);
  if (Date.now() > item.expiry) {
    localStorage.removeItem(key); // Manual eviction on access
    return null;
  }
  return item.value;
};
```

---

3. Interview Scenarios: Eviction Context

Scenario A: "Your app's performance is degrading over time for long-term users."

- **The Problem:** The app is likely hoarding old data in localStorage. Since localStorage is synchronous, the more data the browser has to manage for that origin, the more overhead there is during lookups and initial page loads.
- **The Fix:** Implement a **cleanup script** on app launch that deletes any key older than 7 days.

Scenario B: "How do you handle versioning?"

5. **The Problem:** You changed your data schema (e.g., renamed a field from user_name to username), but the user still has the old object in their localStorage.
6. **The Fix:** Use a SCHEMA_VERSION key. On app load, check if localStorage.VERSION !== CURRENT_VERSION. If they don't match, call localStorage.clear() to prevent the app from crashing due to malformed data.

Summary of Eviction Differences

Feature	Browser HTTP Cache	LocalStorage
Auto-Eviction	Yes (determined by browser)	**No** (Permanent)
Control	Header-based (max-age)	Programmatic (JS)
Strategy	Usually LRU	None (unless you build it)

---

1. The 5MB Limit: Individual or Total?

The 5MB limit applies to the **entire origin**, not per key.

- **Shared Pool:** All keys and values combined for https://example.com must fit within that 5MB (or 10MB, depending on the browser) limit.
- **The Calculation:** The browser calculates the size based on the **UTF-16 strings**. Since UTF-16 uses 2 bytes per character, storing 2.5 million characters will roughly hit a 5MB limit.

---

2. Partial Storage vs. QuotaExceededError

This is a "fail-fast" system. If you try to store 10MB of data into a 5MB slot:

- **No Partial Saves:** The browser will **not** save the first 5MB and discard the rest. The operation is atomic in the sense that if the total size (existing data + new data) exceeds the quota, the **entire setItem call fails**.
- **The Result:** JavaScript will immediately throw a DOMException labeled QuotaExceededError.
- **Existing Data:** Your previously saved data remains untouched and safe; only the new

write is rejected.

---

3. Interview "Deep Dive" Questions

"How do you handle a QuotaExceededError in a production app?"

In a high-level design, you shouldn't just let the app crash. You should propose a **storage fallback strategy**:

- **Catch the Error:** Use a try...catch block around every setItem.
- **Eviction (LRU):** As we discussed, if the catch block triggers, you programmatically delete the oldest or least important keys and retry the save.
- **Graceful Degradation:** If eviction doesn't free enough space (or you've deleted everything you can), fall back to **In-Memory storage** (a simple JS object). Notify the user that "Settings cannot be saved for this session" if it's a critical failure.
- **Migration:** If your app consistently hits this limit, it is a design signal to migrate that specific data type to **IndexedDB**.

"Does every tab get its own 5MB?"

No. All tabs/windows open to the same origin share the same 5MB bucket. If Tab A fills up 4MB, Tab B only has 1MB left to work with. This is why cross-tab coordination (using the storage event) is important to ensure one tab doesn't "starve" the others of storage space.

Summary Table

Scenario	Result
Storing 10MB total	QuotaExceededError thrown; 0 bytes of that 10MB stored.
Two tabs writing simultaneously	Last writer wins; total limit shared between both.
Storing a 1MB string	Works fine (assuming < 4MB already exists).
Storing an Image (Base64)	Very risky; images quickly eat up the 5MB string quota.