In a Frontend System Design interview, **Schema Versioning** is the solution to the "Breaking Change" problem. When you deploy a new version of your app that expects a different data structure, users with "old" data in their localStorage or sessionStorage will experience crashes or bugs.

Here is how you design a robust versioning system.

---

1. The Strategy: The "Wrapper" Pattern

Instead of storing raw data, you store an "Envelope" or "Package" that includes a version number.

Implementation Example

```
const CURRENT_SCHEMA_VERSION = 2.0;

const saveToStorage = (key, data) => {
  const envelope = {
    version: CURRENT_SCHEMA_VERSION,
    payload: data,
    timestamp: Date.now()
  };
  localStorage.setItem(key, JSON.stringify(envelope));
};

const getFromStorage = (key) => {
  const rawData = localStorage.getItem(key);
  if (!rawData) return null;

  try {
    const envelope = JSON.parse(rawData);

    // Version Check Logic
    if (envelope.version < CURRENT_SCHEMA_VERSION) {
      console.warn("Outdated schema detected. Handling migration...");
      return migrateData(envelope);
    }

    return envelope.payload;
  } catch (e) {
    console.error("Malformed JSON in storage");
```

```
    return null;
  }
};
```

---

2. Handling the Mismatch: Two Approaches

When the versions don't match, you have two architectural choices to present to your interviewer:

A. The "Nuclear" Option (Simple & Safe)

If the data is just a cache (e.g., a list of products), simply delete it and fetch fresh data.

- **Pros:** Guaranteed stability; no complex migration code.
- **Cons:** Loss of user preferences or draft data.

B. The Migration Pipeline (Complex & User-Friendly)

If the data is valuable (e.g., a half-finished form), you write a transformation function.

```
const migrateData = (oldEnvelope) => {
  let data = oldEnvelope.payload;

  // Incremental migration: v1 -> v2
  if (oldEnvelope.version === 1.0) {
    // Example: change 'name' string to an object { first, last }
    data = {
      firstName: data.name.split(' ')[0],
      lastName: data.name.split(' ')[1] || ''
    };
  }

  // Save the updated version back to storage
  saveToStorage('user_profile', data);
  return data;
};
```

---

3. Interview Deep-Dive: Global Versioning

Sometimes you don't want to check versioning key-by-key. You might propose a **Global App Version**.

- **Logic:** On app initialization (e.g., in a useEffect or main.js), check a single key: APP_STORAGE_VERSION.
- **Action:** If the stored version is 1.0 and your current code is 2.0, you call localStorage.clear() or sessionStorage.clear() immediately to "reset" the client state before the components even mount.

---

4. Why this is "Must-Know" for SD Interviews

1. **Data Integrity:** It proves you understand that frontend code and frontend "databases" are decoupled in time (Old data meets New code).
2. **Graceful Degradation:** Instead of a TypeError: cannot read property 'split' of undefined, the app handles the discrepancy behind the scenes.
3. **Analytics/Monitoring:** In a real-world system, you would log how many users are hitting "Migration" paths to see how fast your user base is adopting the new version.
