In a Frontend System Design interview, **API Caching** is the bridge between "Network Requests" and "Client Storage." It focuses on how we store the **data** returned by our servers to avoid redundant network calls.

This is different from HTTP Caching (which is automatic) because here, **you** write the logic to manage the data.

---

1. Where does API Caching live?

There are two main "containers" for API cache:

1. **Memory Cache (Store):** Using state management libraries like **TanStack Query (React Query)**, **SWR**, or **Redux**.
2. **Persistent Cache:** Moving that API data into **IndexedDB** so it survives a page refresh or offline state.

---

2. The Core Pattern: Stale-While-Revalidate (SWR)

This is the "Gold Standard" for API caching in modern frontend apps.

- **The Flow:**
    1. User opens a page.
    2. The app immediately shows **stale data** from the cache (Instant UI).
    3. The app fetches **fresh data** from the API in the background.
    4. The app updates the UI and the cache with the fresh data.

Why it's needed: It solves the "Loading Spinner" problem. Users see content immediately, even if it's slightly old, while the app synchronizes.

---

3. Cache Invalidation Strategies

Storing data is easy; the hard part is knowing when to throw it away. In an interview, mention these three:

- **Time-Based (TTL):** "This product list is valid for 5 minutes." After that, the next request must go to the network.
- **Manual Invalidation (Tags/Keys):** When a user updates their profile, you "bust" or "invalidate" the ['user-profile'] cache key.
- **Version-Based:** If the API schema changes, clear the entire cache to prevent the app from crashing.

4. Architectural Deep Dive: Normalization

If you cache raw API responses, you might end up with **duplicate data**.

- **The Problem:** An "Author" object might appear in 10 different "Post" API responses. If the author changes their name, you have to update 10 different cache entries.
- **The Solution: Normalize** the cache. Store authors in an authors table and posts in a posts table, referencing the author by ID. Libraries like **Apollo Client** (for GraphQL) do this automatically.

---

5. Security & Privacy

- **Sensitive Data:** Never cache PII (Personally Identifiable Information) or sensitive financial data in a way that is accessible to other users on a shared machine.
- **Session-Bound:** Ensure that when a user **logs out**, the API cache is wiped (cache.clear()). If you forget this, the next person to use the computer might see the previous user's data.

---

6. Most Asked Interview Q&A

Q: "How do you handle 'Race Conditions' in API caching?"
A: Use **Request Cancellation**. If a user clicks "Category A" then quickly clicks "Category B," you should cancel the "Category A" request and ignore its response so it doesn't overwrite the newer "Category B" data in the cache.

Q: "What is 'Optimistic UI' in the context of caching?"
A: When a user likes a post, you manually update the API Cache to show "Liked" immediately. If the server request fails later, you "roll back" the cache to the previous state and show an error.

Q: "How do you prevent 'Cache Stampede'?"
A: Use **Request Deduplication**. If three different components on a page all need the current-user data at the same time, the caching layer should only send one network request and share the result with all three.

Summary Comparison

Strategy	Speed	Freshness	Complexity
Network Only	Slow	100%	Low
Cache First	Instant	Low (Stale)	Medium
Stale-While-Revalidate	Instant	High (Eventually)	High

---

1. The Professional "Vanilla" Implementation

A robust implementation requires a **Cache Map** and an **In-Flight Map** to handle request deduplication.

```
class ApiCache {
 constructor(ttl = 60000) {
  this.cache = new Map();      // Stores { data, expiry }
  this.inFlight = new Map();   // Stores active Promises
  this.ttl = ttl;
 }

 async fetch(url) {
  // 1. Check if we have valid data in cache
  if (this.cache.has(url)) {
   const { data, expiry } = this.cache.get(url);
   if (Date.now() < expiry) {
    return data; // Cache Hit
   }
   this.cache.delete(url); // Evict expired data
  }

  // 2. Check if the same request is already "In-Flight"
  if (this.inFlight.has(url)) {
   return this.inFlight.get(url); // Return the existing promise (Deduplication)
  }

  // 3. Perform the actual fetch
  const promise = fetch(url)
   .then(res => {
    if (!res.ok) throw new Error("Network error");
    return res.json();
   })
   .then(data => {
    // Store in cache with expiry
    this.cache.set(url, {
     data,
     expiry: Date.now() + this.ttl
    });
    return data;
   })
```

```
      .finally(() => {
        // Clean up in-flight map
        this.inFlight.delete(url);
      });

    this.inFlight.set(url, promise);
    return promise;
  }
}
```

---

2. Key Senior Concepts in this Code

A. Request Deduplication (The inFlight Map)

If five components on your page call api.fetch('/user') at the exact same time, this code ensures only **one** network request is sent. The other four components "subscribe" to the first component's Promise.

B. Handling Race Conditions

By using .finally() to delete the inFlight entry, we ensure that if a request fails, subsequent calls have a chance to try again. Without this, a single failed promise would "clog" the system, and no one would ever be able to fetch that URL again until a page reload.

C. Explicit TTL (Time to Live)

Unlike a simple memoization function, this version respects time. You must explain to the interviewer that **stale data is worse than no data**. By checking Date.now() < expiry, we guarantee the user doesn't see data from an hour ago.

---

3. Scaling the Vanilla Solution (The "Design" Part)

If you are building a large system, you need to add these three features to your Vanilla implementation:

- Garbage Collection (LRU): If the cache Map grows too large, it will cause a memory leak.
  - **The Fix:** Every time you set, check cache.size. If it's over 100, delete the oldest entry.

- **Persistent Fallback:**
    - **The Fix:** In the .then() block, in addition to saving to the Map, save the result to localStorage or IndexedDB. On app startup, "hydrate" the Map from storage.
- **AbortController Integration:**
  If a user navigates away from a page, you should cancel the "In-Flight" request to save bandwidth.
    - **The Fix:** Pass a signal to the fetch method.

---

4. Interview Q&A

Q: "Why use a Class instead of a global object?"
A: Encapsulation. It allows us to create different cache instances with different TTLs (e.g., userCache with 1 hour TTL vs. stockPriceCache with 5 second TTL).

Q: "What happens if the API response is a 500 error? Do you cache that?"
A: No. In the code above, the .then() only caches if res.ok is true. Caching an error (called Negative Caching) is usually a bad idea unless you specifically want to prevent "hitting a broken server" too frequently.

Q: "How do we handle unique requests (e.g., same URL but different Auth headers)?"
A: The url alone is no longer a unique key. You must generate a Cache Key by hashing the URL + Headers + Body.
