```
                while (section.firstElementChild) {
                    section.firstElementChild.remove();
                }
                //add the pocket to DOM
                section.append(pocket);
            }
            //fill pockets with templates
            if(autoFill) core.pk.soc();
        }
    }
  })(),
}
})()
```

Thought for 35s

Analyzing code structure

- The provided JS file defines a core module with version '20251112.0' and counters for 'be', 'cr', and 'pk' elements.
- It's structured as a large immediately invoked function expression (IIFE) that returns an object called `core` with sub-objects like `be` for backend, `cb` for callbacks, `cr` for create, `hf` for helpers, `pk` for pocket, `md` for modular, `sv` for validation, `ud` for user-defined, and `ux` for user experience.

Core.js Implementation Documentation

Overview

Core.js is a lightweight, modular JavaScript framework designed for building dynamic, data-driven web applications. It provides tools for fetching data from backend sources, managing templates, rendering content using placeholders and cloning mechanisms, handling user interactions, validation, formatting, and more. The framework is structured as an immediately invoked function expression (IIFE) that exposes a single global object `core`, which encapsulates all functionality.

Key features include:

- **Data Fetching and Caching**: Asynchronous fetching of JSON data and HTML templates with caching mechanisms.
- **Templating and Rendering**: Supports placeholder injection (e.g., `{{data:user:name}}`), cloning for lists, and dynamic DOM manipulation.
- **Validation and Formatting**: Built-in scrubbers for input validation and formatters for values (e.g., dates, currency).
- **User-Defined Hooks**: Extensible via `ud` (user-defined) functions for customization at various lifecycle points.
- **Routing and State Management**: Optional URL hash-based routing for managing page states.
- **Debugging and Performance**: Logging, timeouts, and performance tracking.

The framework initializes automatically on DOMContentLoaded or immediately if the document is ready. It supports local or CDN-based installations and checks for a local setup via `/module/install.json`.

Version: Defined as `core_version = '20251112.0'`.

Counters: Tracks ongoing operations with `core_be_count` (backend fetches), `core_cr_count` (create operations), `core_pk_count` (pocket rendering).

Setup and Initialization

- **Base URL**: Defaults to a CDN (`https://cdn.jsdelivr.net/gh/Sitezip/core.sbs@` + version). Can switch to local origin if `/module/install.json` returns `{success:true}`.
- **User Settings**:
  - `useDebugger`: Boolean (default: false) – Enables console logging.
  - `useRouting`: Boolean (default: false) – Uses URL hash for state management.
  - `useLocking`: Boolean (default: true) – Locks pockets after rendering to prevent refreshes.
- **Initialization**:

- `core.init()` : Called on DOM ready. Initializes `cr` (templates), adds click listeners, runs user-defined `ud.init()` , initializes `pk` (pockets), and fetches internal checks/objects.
- Fetches `/module/install.json` to detect local install.
- Fetches `/core.json` for internal objects (processed in `be.postflight`).

Core Structure

The `core` object is composed of nested objects for organization:

`be` (Backend Functions)

Handles data and template fetching with caching.

- **Properties**:
  - `cacheCreateTs` : Object tracking creation timestamps for data/templates.
  - `cacheExpire` : Object for custom expiration times (e.g., `{type: 'data', name: 'quote', seconds: 5}`).
  - `cacheExpireDefault` : Default expiration (86400 seconds).
  - `fetchLogFIFO` : Logs fetch settings for debugging/retry.
- **Methods**:
  - `setCacheTs(dataRef, type)` : Sets creation timestamp for cache.
  - `checkCacheTs(dataRef, type)` : Checks if cache is valid based on expiration.
  - `setGetParams(settings)` : Builds fetch parameters (method, cache, headers, body). Supports FormData or JSON body.
  - `getData(dataRef, dataSrc, settings)` : Fetches JSON data. Uses cache if valid; otherwise fetches and stores in `cr` . Handles pre/postflight hooks.
  - `getTemplate(dataRef, dataSrc, settings)` : Fetches template text. Similar to `getData` but for strings.
  - `preflight(dataRef, dataSrc, type, settings)` : Prepares settings, applies user-defined `ud.preflight()` , logs to FIFO.

- `postflight(dataRef, dataObj, type)`: Post-processes response. Handles internal objects (e.g., removes 'Use' keys), applies `ud.postflight()`.

`cb` (Callback Functions)

Lifecycle hooks for rendering.

- **Methods**:
  - `prepaint(dataRef, dataObj, type)`: Called before inserting template/data. Invokes `ud.prepaint()`.
  - `postpaint(dataRef, dataObj, type)`: Called after inserting. Invokes `ud.postpaint()`.

`cr` (Create Functions)

Manages data storage and templates in DOM or sessionStorage.

- **Properties**:
  - `storageIdDefault`: Default storage mode (1: dataset, 0: DOM property, 2: sessionStorage).
- **Methods**:
  - `init()`: Preloads templates from `<template name="...">` elements. Sets defaults for 'EMPTY' and 'LOADING'.
  - `delData(name, elem, storageId)`: Deletes data by name from chosen storage.
  - `setData(name, data, elem, storageId)`: Stores JSON-stringified data. Uses sessionStorage for internal ('coreInternal*').
  - `getData(name, elem, storageId)`: Retrieves data, checks cache expiration, triggers refresh if expired.
  - `delTemplate(name)`: Removes template element.
  - `setTemplate(name, value)`: Creates/updates `<template>` with escaped content.
  - `getTemplate(name)`: Retrieves unescaped template, applies `ud.getTemplate()`, injects via `pk.injector()`.

`hf` (Helper Functions)

Utility functions for common tasks.

- **Properties**:
  - `prevSortKey` : Tracks last sort key for reversing.
- **Methods**:
  - `addClickListeners()` : Applies listeners to all `<a>` tags.
  - `addClickListener(element)` : Adds click handler for links with `data-core-templates` or `data-core-data` . Prevents default, inserts pocket.
  - `ccNumAuth(ccNum)` : Validates credit card (Luhn algorithm) and detects type (Visa, etc.).
  - `copy(text)` : Copies text to clipboard using a temporary textarea.
  - `date(dateStr, format, strict)` : Formats dates with tokens (e.g., 'M/D/YY', 'TS' for timestamp). Supports strict mode.
  - `digData(object, ref)` : Deeply accesses object properties via dot/comma notation (e.g., 'addresses.billing.street'). Handles arrays with '[n]'.
  - `parseJSON(str)` : Safe JSON parse, returns undefined on error.
  - `getRoute(which)` : Gets URL part (e.g., 'href', 'hash').
  - `setRoute(base, title, append, info)` : Updates history state and URL.
  - `sortObj(objects, key, type, sort)` : Sorts array of objects by key (numeric/string, ASC/DESC). Reverses on repeated ASC sort.
  - `uuid(prefix, delim)` : Generates UUID.
  - `hydrateByClass(classFilter)` : Hydrates elements by class (e.g., 'h-user-name' → appends value from data). Supports options like 'h--countdown'.
  - `formatByClass(classFilter)` : Formats content by class (e.g., 'f-money' → '$1.00'). Supports persistent 'f--' and data attributes.

`pk` (Pocket Functions)

Manages "pockets" – containers for dynamic content rendering.

- **Properties**:
  - `timeout` : Render timeout (2000ms).

- **Methods**:
  - `init()` : Handles routing from hash, calls `soc()` .
  - `eoc()` : Cleanup, hydrates/formats, calls `ud.eoc()` , updates route if enabled.
  - `soc()` : Starts rendering cycle, waits for backend, calls `ud.soc()` .
  - `getTemplate()` : Fetches required templates, adds 'LOADING', checks completion/timeout.
  - `addTemplate()` : Inserts templates into pockets, hides/shows pockets.
  - `getData()` : Fetches data for clones, checks cache.
  - `addData()` : Clones and injects data into templates, adds listeners.
  - `injector(templateStr)` : Replaces `{{data:...}}` placeholders with data values and formats.
  - `cloner(records, cloneStr)` : Clones template for each record, replaces `{{rec:...}}` or `{{aug:...}}` (e.g., index). Supports `ud.cloneValue/String()` .

`md` (Modular Functions)

Dynamic module loading (e.g., forms).

- **Properties**:
  - `formSubmitLockout` : Prevents multiple submits.
- **Methods**:
  - `form(funcName, args)` : Lazy-loads `/module/form.js` and calls function.

`sv` (Validation Functions)

Input scrubbing and formatting.

- **Properties**:
  - `regex` : Object with common regex patterns (email, phone, etc.).
- **Methods**:
  - `format(value, formatStr, valueDefault)` : Applies formats (e.g., 'money*USD', 'upper'). Extensive list (alphaonly, date, encrypt, etc.).

- `scrub(scrubArr)` : Validates array of objects (e.g., [{name: 'email', value: '...', scrubs: ['req', 'email']}]).
- `scrubEach(scrubObj, scrubArr)` : Validates single object, applies formats.
- `scrubSimple(name, value, scrubs)` : Wrapper for single scrub.
- `scrubMatch(scrubArr, scrubMatch, valueMatch)` : Checks if values match across objects.

`ud` (User-Defined)

Customization points and defaults.

- **Properties** (Getters/Setters):
  - `defaultDelta` : Empty value fallback ('').
  - `defaultDeltaFormat` : Default format ('none').
  - `defaultClickTarget` : Link target ('main').
  - `defaultDateFormat` : 'M/D/YY H:MM P'.
  - `defaultLoadingTemplate` : '<marquee>loading...</marquee>'.
  - `defaultEmptyTemplate` : ''.
  - `defaultPageTitle` : 'core.js'.
  - `defaultPageStatusUpdate` : 'Updated bookmark location'.
  - `alert*` : Various error messages.
  - `hydrationClassIgnoreList` : ['h-100'].
  - `formatClassIgnoreList` : [].
- **Expected User Functions** (Optional):
  - `init()` , `soc()` , `preflight(dataRef, dataSrc, type)` , `postflight(dataRef, dataObj, type)` , `prepaint(...)` , `postpaint(...)` , `eoc()` , `formatValue(value, formatList, clue)` , `getTemplate(name, template)` , `cloneValue(record, args)` , `cloneString(record, args)` .

`ux` (User Experience)

High-level UI utilities.

- **Methods**:
  - `formatValue(value, formatList, clue)` : Applies multiple formats via `sv.format()` , calls `ud.formatValue()` .
  - `insertPocket(target, dataRefs, dataSources, autoFill)` : Creates/inserts pocket element, sets attributes, optionally fills via `pk.soc()` .

Lifecycle and Flow

1. **Init**: Load settings, fetch internal data, init modules.
2. **Click/Route**: Insert pocket via `ux.insertPocket()` .
3. **Render Cycle** (`pk.soc()` → `eoc()`): Fetch templates/data (with cache), insert, clone, hydrate/format.
4. **Backend**: `be.get*` → preflight → fetch → postflight → store in `cr` .
5. **Customization**: Hooks at every stage.

Extensibility and Best Practices

- Extend via `ud` functions.
- Use `data-core-*` attributes for dynamic elements.
- Avoid direct DOM manipulation; rely on pockets/clones.
- For performance, set cache expirations and use locking.
- Debug with `useDebugger: true` .

This documentation covers the core implementation. For usage examples, refer to inline comments or inferred patterns (e.g., placeholders like `{{rec:name:upper}}`). If needed, expand with specific code snippets.

Explain pocket rendering examples

Pocket Rendering in Core.js – Detailed Examples

Pocket rendering is the **core dynamic content mechanism** in `core.js` . It allows you to define **data-driven UI sections** (called "pockets") that are **automatically populated**

with templates and data, using **caching**, **cloning**, **placeholder injection**, and **lifecycle hooks**.

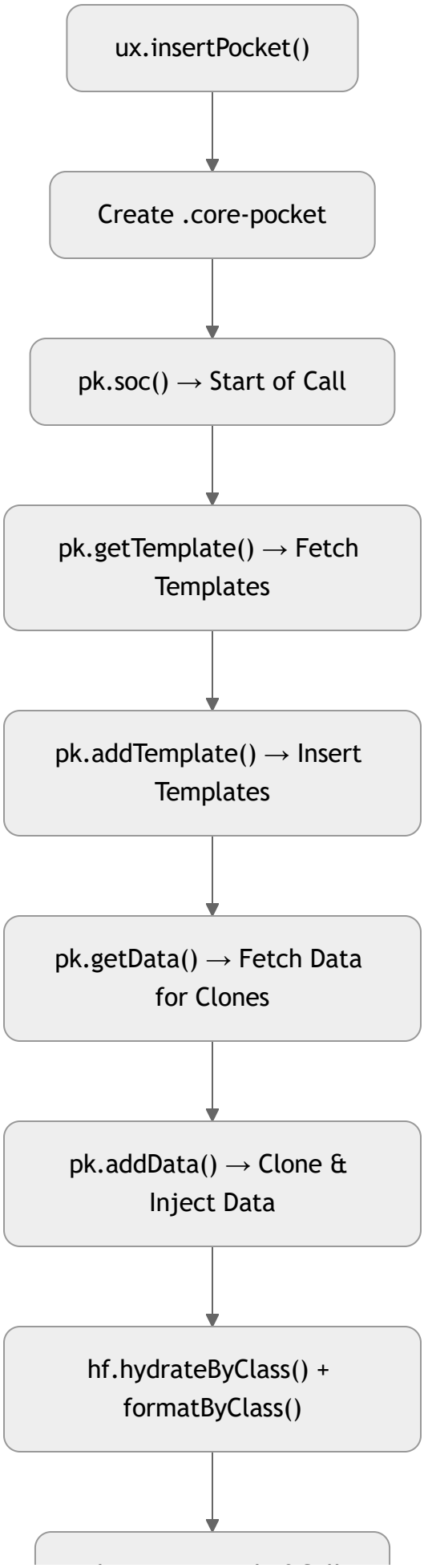
What is a "Pocket"?

A **pocket** is a `div` with the class `core-pocket` that acts as a **container** for dynamic content.

```html
<div class="core-pocket" data-core-templates="user-card" data-user-card-c
```

- `data-core-templates` : Comma-separated list of **template names** to insert.
- `data-{template}-core-source` : URL to fetch data or template (optional).
- The pocket is **filled automatically** when `core.pk.soc()` runs.

How Pocket Rendering Works (Step-by-Step)


ux.insertPocket()
Create .core-pocket
pk.soc() → Start of Call
pk.getTemplate() → Fetch Templates
pk.addTemplate() → Insert Templates
pk.getData() → Fetch Data for Clones
pk.addData() → Clone & Inject Data
hf.hydrateByClass() + formatByClass()

pk.eoc() → End of Call

Example 1: Simple Template Injection

HTML

```html
<div id="main"></div>

<template name="greeting">
  <h1>Hello, {{data:user:name}}!</h1>
  <p>Today is {{data:coreInternalObjects:date:DATE}}.</p>
</template>
```

JavaScript

```js
core.ux.insertPocket('#main', 'greeting');
```

What Happens

1. Pocket created: `<div class="core-pocket" data-core-templates="greeting">`
2. `greeting` template found in DOM → inserted.
3. `{{data:user:name}}` → replaced via `pk.injector()` using `core.cr.getData('user')`.
4. `{{data:coreInternalObjects:date:DATE}}` → uses internal date object.

> ***Note**: `user` data must be preloaded or fetched via `data-user-core-source`.*

Example 2: Data + Template + Cloning

HTML

```html
```

```
<div id="users"></div>

<template name="user-card">
  <div class="core-clone" data-core-data="users" data-core-source="/api/u
    <div class="card">
      <h3>{{rec:name:upperfirst}}</h3>
      <p>Email: {{rec:email}}</p>
      <p>Index: {{aug:i}}</p>
      <p>Count: {{aug:c}}</p>
    </div>
  </div>
</template>
```

JavaScript

```
js

core.ux.insertPocket('#users', 'user-card');
```

What Happens

1. Pocket created.
2. `user-card` template inserted.
3. `core-clone` found → triggers `pk.getData('users')`.
4. Data fetched from `/api/users.json` → stored in `cr` as `users`.
5. `pk.cloner()` runs:
   - Loops through each user.
   - Clones the `.card` for each.
   - Replaces:
     - `{{rec:name:upperfirst}}` → "John Doe" → "John doe"
     - `{{rec:email}}` → "john@example.com"
     - `{{aug:i}}` → 0, 1, 2...
     - `{{aug:c}}` → 1, 2, 3...
6. Cloned elements get class `core-cloned-users-*`.

7. `coreRecord` stored on each clone via `storageId=0`.

Example 3: Click-to-Load Pocket (Dynamic)

HTML

```html
<a href="#"
   data-core-templates="quote"
   data-quote-core-source="/api/quote.json"
   target="quote-section">
  Get Random Quote
</a>

<div id="quote-section"></div>

<template name="quote">
  <blockquote>
    "{{data:quote:text}}"
    <footer>— {{data:quote:author}}</footer>
  </blockquote>
</template>
```

What Happens on Click

1. `hf.addClickListener()` → prevents default.
2. `ux.insertPocket('#quote-section', 'quote', [{name:'quote', url:'/api/quote.json'}])`
3. Pocket created.
4. `quote` template inserted.
5. `quote` data fetched → `{{data:quote:text}}` replaced.

> ***Result**: New quote appears every click.*

Example 4: Hydration & Formatting by Class

HTML

```html
<div id="profile"></div>

<template name="profile">
  <div>
    <h1 class="h-user-name">Loading...</h1>
    <p class="h-user-email f-lower">...</p>
    <p class="f-money" data-f-clue="$">0</p>
    <p class="f-date-time" data-f-clue="M/D/YY">0</p>
  </div>
</template>
```

JavaScript

```js
// Preload user data
core.be.getData('user', '/api/user.json');

// Later...
core.ux.insertPocket('#profile', 'profile');
```

What Happens

1. Template inserted.
2. `pk.eoc()` → `hf.hydrateByClass()`:
   - `h-user-name` → "John Doe" (appended, class removed)
   - `h-user-email` → "JOHN@EXAMPLE.COM"
3. `hf.formatByClass()`:
   - `f-money` → "$1,234.56"
   - `f-date-time` → "11/13/25"

> *Use `h--` or `f--` for **persistent** hydration/formatting.*

Example 5: Routing with Hash (Bookmarkable Pockets)

Enable routing:

```js
core.useRouting = true;
```

Click a link

```html
<a href="#" data-core-templates="user,posts" target="main"
   data-user-core-source="/user/123.json"
   data-posts-core-source="/user/123/posts.json">
  View Profile
</a>
```

URL becomes

```text
https://yoursite.com/#%7B%22t%22%3A%22%23main%22%2C%22l%22%3A%5B%7B%22n%2
```

On page load

- `pk.init()` reads hash → rebuilds exact same UI.
- **Bookmarkable, shareable, reload-safe**.

Example 6: Silent Preload (No Pocket)

```js
core.ux.insertPocket('core_be_getData:user', 'user', [{name:'user', url:'
```

- No DOM pocket created.
- Only fetches and caches `user` data.
- Useful for preloading.

Example 7: Multiple Templates in One Pocket

```html
<template name="header">...</template>
<template name="list">...</template>
<template name="footer">...</template>
```

```js
core.ux.insertPocket('#main', 'header,list,footer');
```

→ All three templates inserted **in order**.

Example 8: Custom Cloning Logic

```js
core.ud.cloneValue = function(record, args) {
  if (args.str1 === 'status') {
    return record.active ? 'Active' : 'Inactive';
  }
};
```

```html
<template name="user-row">
  <tr class="core-clone" data-core-data="users">
    <td>{{rec:name}}</td>
    <td>{{aug:v:status}}</td>
  </tr>
</template>
```

→ `{{aug:v:status}}` → "Active"

Summary: Pocket Rendering Flow

Step	Function	Purpose
1	`ux.insertPocket()`	Creates pocket
2	`pk.soc()`	Starts render cycle
3	`pk.getTemplate()`	Fetches missing templates
4	`pk.addTemplate()`	Inserts templates
5	`pk.getData()`	Fetches data for clones
6	`pk.addData()`	Clones + injects data
7	`hf.hydrateByClass()`	Fills `h-*` classes
8	`hf.formatByClass()`	Formats `f-*` classes
9	`pk.eoc()`	Cleanup, routing, `ud.eoc()`

Best Practices

Do	Don't
Use `data-core-templates` + `data-*-core-source`	Manipulate DOM directly
Preload common data	Rely on stale cache
Use `useLocking = true` for static pages	Forget `core.pk.soc()` after dynamic insert
Use `h--` / `f--` for live updates	Overuse `setTimeout`

Debug Tips

```js
core.useDebugger = true;
```

Logs:

- Cache hits/misses
- Fetch timing
- Template/data loading
- Performance (`X.XXms`)

Pocket rendering = declarative, cache-aware, extensible UI updates.
Use it for lists, modals, dashboards, SPAs — anywhere dynamic content lives.

Template caching strategies

Template Caching Strategies in Core.js

Core.js provides a **robust, multi-layered caching system** for both **data** and **templates**, with fine-grained control over **expiration**, **storage**, and **refresh behavior**. This document focuses specifically on **template caching** — how templates are fetched, stored, retrieved, and refreshed.

---

1. Template Storage & Retrieval

Templates are stored in the DOM as `<template>` elements under a container (`#cr-data` or cloned `template`).

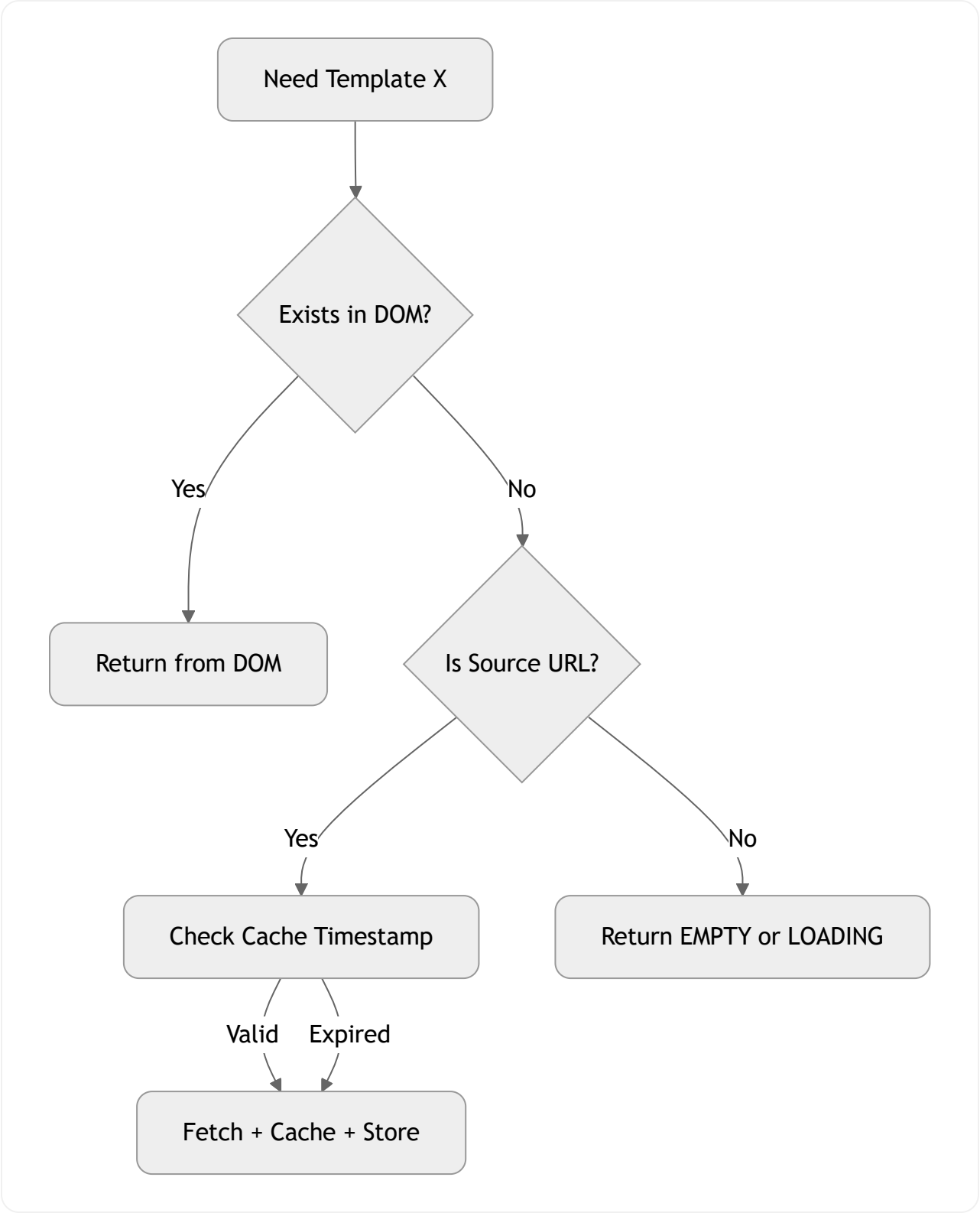
```html
<template name="user-card">...</template>
```

Storage Methods (`cr.setTemplate`)

Method	Storage	Use Case
`DOM`	`<template name="...">` in `#cr-data`	Default, fast, visible in DOM
`sessionStorage`	JSON string	Fallback if DOM not available
`Custom via` ud.getTemplate() ` **	Any	Override storage

> ***Default**: Templates are **appended to** `#cr-data` (or cloned `template` if not found).*

2. Template Fetching & Caching Flow



3. Cache Control Settings

Global Settings (via `core.be`)

```
js
```

```
core.be.cacheExpireDefault = 3600; // 1 hour default
core.be.cacheExpire = {type:'template', name:'user-card', seconds: 300};
```

Per-Template Override

```html
<!-- Fetch with 5-minute cache -->
<div class="core-pocket"
     data-core-templates="quote"
     data-quote-core-source="/api/quote.html?cache=300">
</div>
```

> *`core.be.preflight()` can read URL params or `data-*` to set per-request cache.*

---

4. Cache Key & Timestamp Logic

- **Key**: `dataRef` (template name, e.g., `"user-card"`)
- **Type**: `'template'`
- **Timestamp**: Stored in `core.be.cacheCreateTs.template[dataRef]`
- **Expiration**: `cacheExpire.template[dataRef]` or `cacheExpireDefault`

```js
core.be.checkCacheTs('user-card', 'template') // → true/false
```

---

5. Caching Strategies

Strategy 1: Aggressive Caching (Default)

```js
core.be.cacheExpireDefault = 86400; // 24 hours
```

Use Case: Static UI components (headers, footers, card layouts).

Pros:

- Zero network after first load
- Instant rendering

Cons:

- Stale templates until page reload

Strategy 2: Short-Lived Cache (Live Data)

```
js

core.be.cacheExpire = {type:'template', name:'live-feed', seconds: 60};
```

Use Case: Real-time dashboards, live quotes.

Pros:

- Fresh content
- Controlled refresh

Cons:

- Frequent fetches

Strategy 3: Versioned Templates (Recommended)

Append version to URL or name:

```
html

<template name="user-card-v2">...</template>
```

Or via URL:

```
js

core.be.getTemplate('user-card', '/templates/user-card.html?v=2');
```

Use Case: Deploying UI updates without breaking cache.

Pros:

- Full control over cache busting
- Safe deploys

Strategy 4: Conditional Caching via `ud.preflight()`

```js
core.ud.preflight = function(dataRef, dataSrc, type) {
  if (type === 'template' && dataRef === 'promo-banner') {
    // Cache only on weekdays
    const isWeekend = new Date().getDay() % 6 === 0;
    return { cache: isWeekend ? 'no-cache' : 'default' };
  }
};
```

Use Case: Business logic-driven caching.

Strategy 5: Preload + Lock

```js
// Preload critical templates
core.be.getTemplate('header', '/templates/header.html');
core.be.getTemplate('footer', '/templates/footer.html');

// Lock pockets after render
core.useLocking = true;
```

Result: After first render, pockets become `core-pocketed` → **no refresh on** `soc()` .

Use Case: Static pages, SPAs with fixed layout.

Strategy 6: Fallback Templates

```js
core.ud.defaultEmptyTemplate = '<p>No data available</p>';
```

```
core.ud.defaultLoadingTemplate = '<div class="spinner">Loading...</div>';
```

Used when:

- Template not found
- Data empty
- Fetch in progress

---

6. Cache Invalidation Techniques

Method	How to Trigger
Page Reload	Full cache reset
Versioned Name/URL	`user-card-v2`
Manual Clear	`core.cr.delTemplate('name')`
Time-Based	`cacheExpire`
Event-Based	`core.pk.soc()` after data update

```
js

// Force refresh
core.cr.delTemplate('user-card');
core.pk.soc();
```

---

7. Performance Monitoring

Enable debugging:

```
js

core.useDebugger = true;
```

Logs:

```text
core.js cache 'user-card' has expired
core.js template 'user-card' requested
core.js completed in 245.1ms
```

Use `core.hf.date(null, 'perf')` for manual timing.

8. Best Practices Summary

Practice	Recommendation
Cache Static Templates	24h+ or versioned
Cache Dynamic Templates	1–5 min
Preload Critical UI	Headers, nav, card shells
Use `useLocking = true`	For static content
Version Templates	`name-v2`, `?v=2`
Monitor with `useDebugger`	Catch stale/missing templates
Fallbacks	Always define `EMPTY` / `LOADING`

9. Example: Optimized Template Strategy

```html
<!-- Preloaded in <head> -->
<template name="header">...</template>
<template name="footer">...</template>

<!-- Versioned, cached 1 week -->
<template name="product-card-v3">...</template>

<!-- Dynamic, cached 1 min -->
<div class="core-pocket"
     data-core-templates="live-price"
```

```
    data-live-price-core-source="/api/price.html?cache=60">
</div>
```

```js
// Init
core.be.cacheExpireDefault = 86400;
core.be.cacheExpire = {type:'template', name:'live-price', seconds: 60};
core.useLocking = true;
core.useDebugger = true;
```

Conclusion

Core.js template caching is **powerful, predictable, and extensible**:

- **DOM-based** → fast, debuggable
- **Timestamp-driven** → precise control
- **Hookable** → `ud.preflight()` for logic
- **Lockable** → prevent unnecessary re-renders
- **Fallback-safe** → never break UI

Use **versioning + locking** for production.
Use **short cache + `soc()`** for live data.

Master template caching → build blazing-fast, reliable SPAs.

Data caching strategies

Data Caching Strategies in Core.js

Core.js provides a **sophisticated, multi-layered data caching system** that balances **performance**, **freshness**, and **developer control**. This document covers **data-specific** caching (not templates), including storage, expiration, invalidation, and advanced strategies.

---

1. Data Storage & Retrieval

Data is stored using `core.cr.setData()` and retrieved via `core.cr.getData()`.

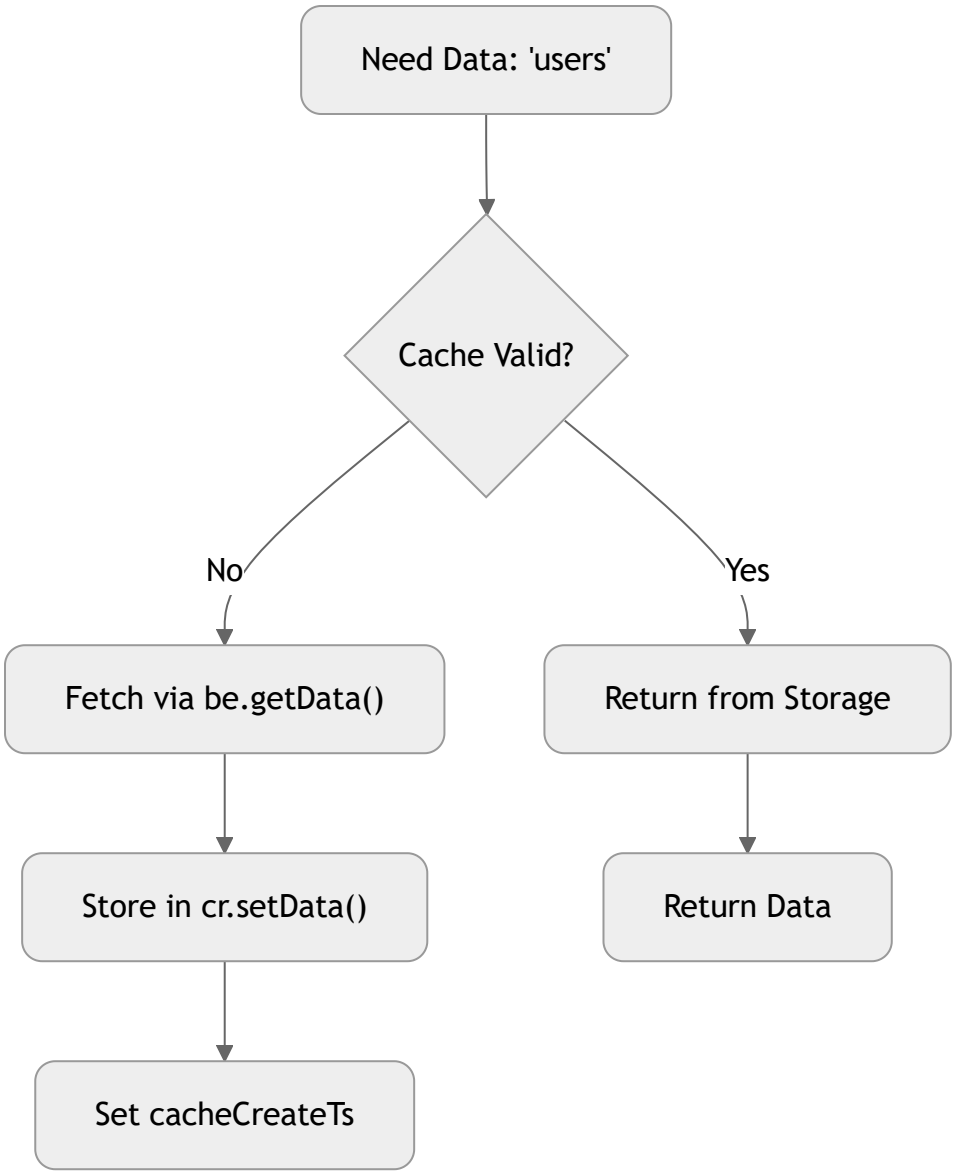
Storage Options (`storageId`)

ID	Method	Storage	Use Case
0	DOM	`element._CORE_Data`	Per-element record (e.g., cloned rows)
1	Dataset	`element.dataset[dataRef]`	Pocket-level data
2	`sessionStorage`	`sessionStorage.setItem()`	Global, persistent across tabs

> ***Default***: *`storageIdDefault = 1` (dataset)*

```js
core.cr.setData('user', {name: 'John'}, document.getElementById('main'),
```

2. Data Fetching & Caching Flow



3. Cache Control Settings

Global Settings

```
js

core.be.cacheExpireDefault = 3600; // 1 hour
core.be.cacheExpire = {type:'data', name:'user', seconds: 300};
```

Per-Request Override

```
js
```

```
core.be.getData('quote', '/api/quote.json', {
  cache: 'no-cache', // force refresh
  // or use preflight
});
```

---

4. Cache Key & Timestamp Logic

- **Key**: `dataRef` (e.g., `"users"`, `"user-123"`)
- **Type**: `'data'`
- **Timestamp**: `core.be.cacheCreateTs.data[dataRef]`
- **Expiration**: `cacheExpire.data[dataRef]` or `cacheExpireDefault`

```
js

core.be.checkCacheTs('users', 'data') // → true if not expired
```

---

5. Caching Strategies

Strategy 1: Aggressive Caching (Static Data)

```
js

core.be.cacheExpireDefault = 86400; // 24 hours
```

Use Case: User profile, settings, static lists.

Pros:

- Instant UI after first load
- Minimal network

Cons:

- Stale until page reload

Strategy 2: Short-Lived Cache (Live Data)

```js
core.be.cacheExpire = {type:'data', name:'live-price', seconds: 10};
```

Use Case: Stock prices, live scores, notifications.

Pros:

- Near real-time
- Controlled staleness

Cons:

- Frequent fetches

Strategy 3: Versioned Data (Cache Busting)

```js
core.be.getData('products-v2', '/api/products.json?v=2');
```

Use Case: Deploying API changes.

Pros:

- Full control
- Safe updates

Strategy 4: Conditional Caching via `ud.preflight()`

```js
core.ud.preflight = function(dataRef, dataSrc, type) {
  if (type === 'data' && dataRef === 'dashboard') {
    const hour = new Date().getHours();
    return {
      cache: (hour >= 9 && hour <= 17) ? 'default' : 'no-cache'
    };
  }
};
```

Use Case: Business hours caching.

Strategy 5: Preload + Lock

```js
// Preload critical data
core.be.getData('user', '/api/user.json');
core.be.getData('settings', '/api/settings.json');

// Lock pockets
core.useLocking = true;
```

Result: After render, pockets become `core-pocketed` → **no refresh**.

Use Case: Dashboard, profile page.

Strategy 6: Per-Element Caching (`storageId=0`)

```js
// In pk.addData()
core.cr.setData('coreRecord', record, newClone, 0);
```

Use Case: Each cloned row has its own data.

Pros:

- No global pollution
- Easy access: `element._CORE_Data.coreRecord`

Strategy 7: Silent Preload (No UI)

```js
core.ux.insertPocket('core_be_getData:user', '', [{name:'user', url:'/api
```

- Fetches and caches `user` data **without rendering**.

- Ideal for pre-warming cache.

6. Cache Invalidation Techniques

Method	How to Trigger
Time-Based	`cacheExpire`
Manual	`core.cr.delData('users')`
Versioned Key	`'users-v2'`
Event-Driven	Call `core.pk.soc()` after update
URL Param	`?t=1731439200`

```js
// Force refresh
core.cr.delData('users');
core.pk.soc();
```

7. Advanced: Cache + Refresh on Demand

```js
// Button to refresh
document.getElementById('refresh').addEventListener('click', () => {
  core.cr.delData('quotes');
  core.pk.soc(); // re-renders pocket
});
```

8. Performance Monitoring

```js
core.useDebugger = true;
```

Logs:

```text
core.js cache 'users' has expired
core.js data 'users' requested
core.js loaded data (1) in 145.2ms
```

9. Best Practices Summary

Practice	Recommendation
Cache User Data	1–24 hours
Cache Live Data	5–60 seconds
Preload on Init	Critical paths
Use `storageId=0`	For cloned records
Use `useLocking = true`	Static content
Version Data Keys	On API changes
Monitor with Debugger	Catch stale data

10. Example: Optimized Data Strategy

```js
// Init
core.be.cacheExpireDefault = 3600;
core.be.cacheExpire = {type:'data', name:'live-price', seconds: 15};
core.useLocking = true;

// Preload
core.be.getData('user', '/api/user.json');
core.be.getData('nav', '/api/nav.json');
```

```
// Live data
core.ux.insertPocket('#price', 'price', [{name:'live-price', url:'/api/pr
```

```html
<!-- Pocket with live data -->
<div id="price" class="core-pocket"
     data-core-templates="price"
     data-live-price-core-source="/api/price.json">
</div>
```

Conclusion

Core.js data caching is **predictable, extensible, and performant**:

- **Multi-storage** → DOM, dataset, session
- **Timestamp-based** → precise expiration
- **Hookable** → `ud.preflight()` for logic
- **Lockable** → prevent re-fetches
- **Preloadable** → instant UI

Use long cache + locking for static data.
Use short cache + `soc()` for live data.

Master data caching → build fast, reliable, real-time apps.

↳ Cache invalidation techniques

↳ Core.js error handling

↳ More concise examples