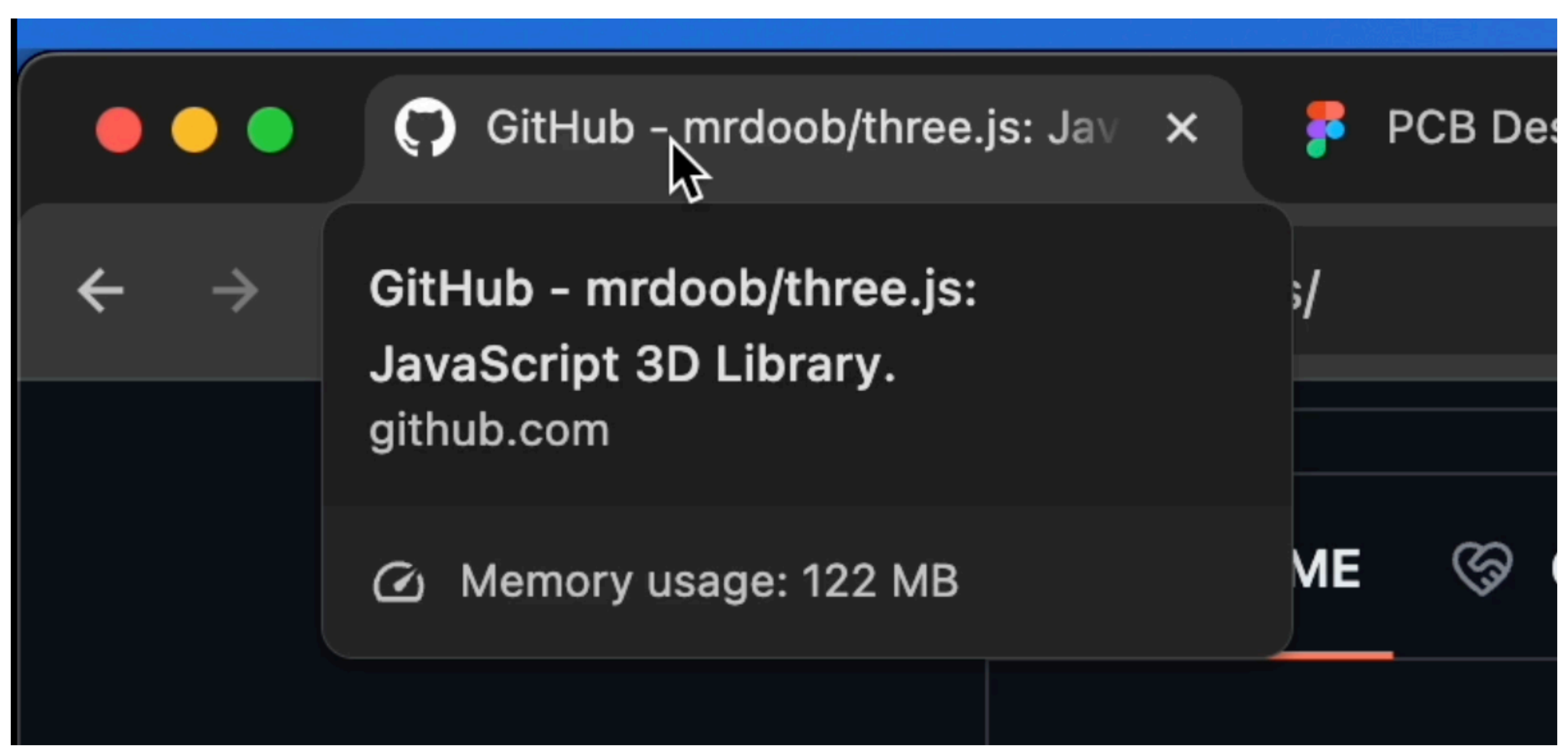
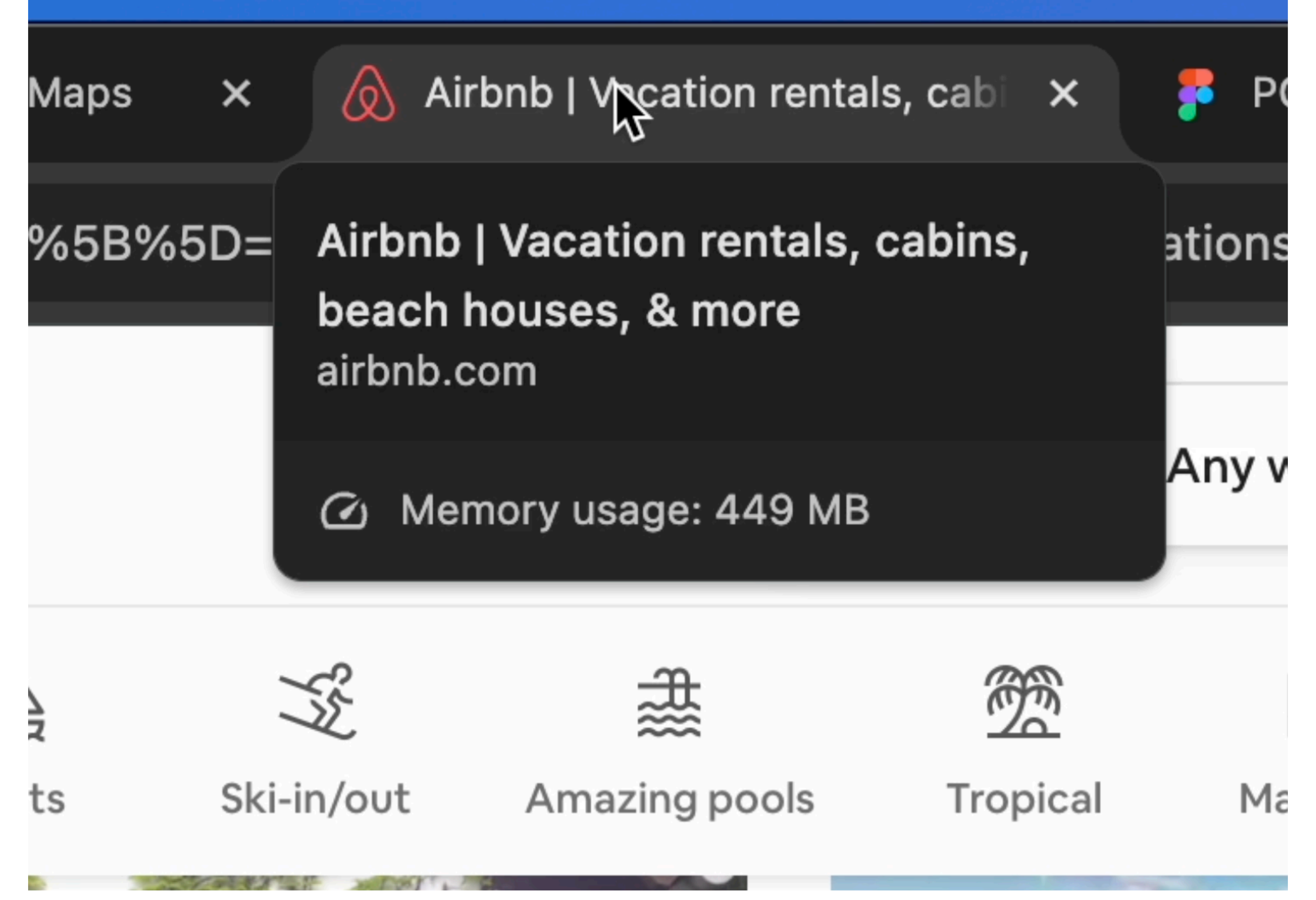
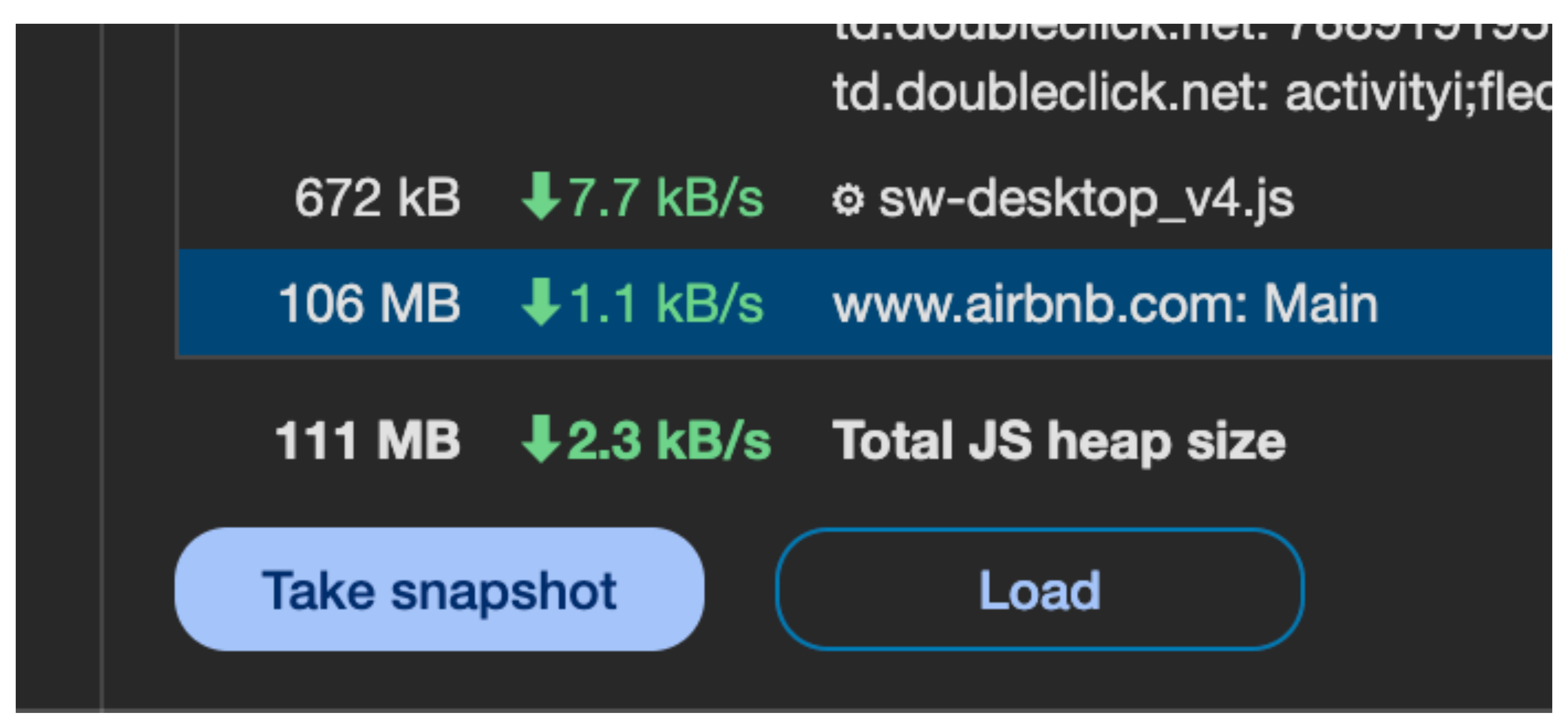
Your web app is taking up **too much RAM**

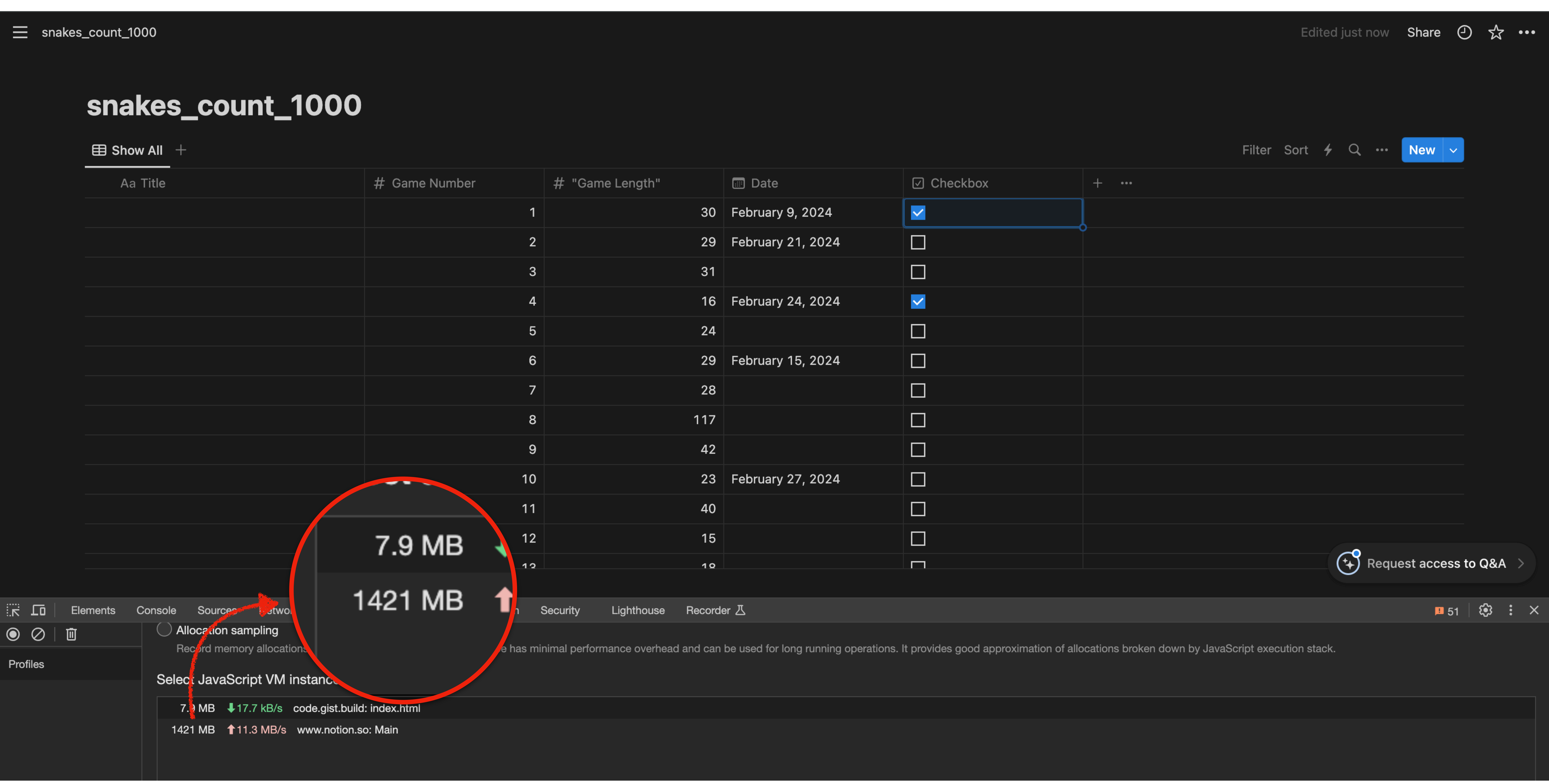
Let's fix it!


GitHub - mrdoob/three.js: Jav
PCB Des
GitHub - mrdoob/three.js:
JavaScript 3D Library.
github.com
Memory usage: 122 MB
ME


Maps
Airbnb | Vacation rentals, cabi
Airbnb | Vacation rentals, cabins, beach houses, & more
airbnb.com
Memory usage: 449 MB
Any v
Ski-in/out
Amazing pools
Tropical



How much of that is actually JS data?



snakes_count_1000
Edited just now
Share
snakes_count_1000
Show All
Filter
Sort
New
Aa Title
Game Number
"Game Length"
Date
Checkbox
1
30
February 9, 2024
2
29
February 21, 2024
3
31
4
16
February 24, 2024
5
24
6
29
February 15, 2024
7
28
8
117
9
42
10
23
February 27, 2024
11
40
12
15
7.9 MB
1421 MB
Request access to Q&A
Elements
Console
Security
Lighthouse
Recorder
51
Allocation sampling
Profiles
has minimal performance overhead and can be used for long running operations. It provides good approximation of allocations broken down by JavaScript execution stack.
17.7 kB/s
code.gist.build: index.html
1421 MB
11.3 MB/s
www.notion.so: Main

LIBRARY OBJECTS RULES

Designator, part name or selecti

- Root
  - Layout
    - Flux.ai
    - Line Shape
    - 12V
    - Flux.ai
    - DESIGNED WITH
    - Line Shape
    - GND
    - Line Shape
    - Flux.ai
    - Components
    - Nets
    - Line Shape
    - Flux.ai

Strobe Flasher Project

Description

High Brightness Strobe/Flashing LED lights. Good for cyclists, e-bikes who bike at night.

Created

November 18th 2022

Last updated by jharwinbarrozo

15 minutes ago

1 Contributor(s)

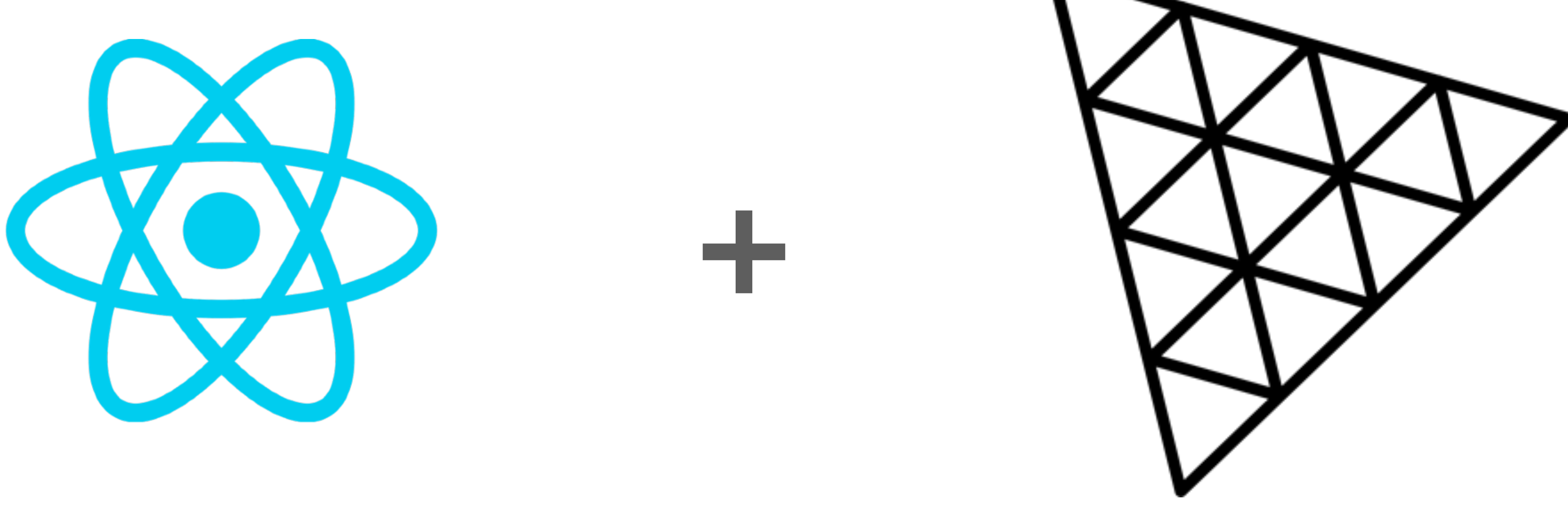
Properties

Availability & Pricing

Add a Manufacturer Part Number to see inline results.

Mouser VISIT SITE

Digi-Key VISIT SITE



jharwinbarrozo/16 channel relay module

Loaded

1 0

Schematic | Schematic NEW | Code | PCB

SHARE

LIBRARY OBJECTS RULES

Designator, part name or selecto

Root

Layout

Top

Layer

65 0

2D

16 channel relay module

Description

16 channel relay that will allow you to control different types of electrical loads

Created

June 17th 2022

Last updated by jharwinbarrozo

3 months ago

1 Contributor(s)

Properties

Availability & Pricing

Add a Manufacturer Part Number to see inline results.

Mouser	VISIT SITE
Digi-Key	VISIT SITE
LCSC	VISIT SITE

Is faster enough?

1546mb /4096
Memory
5

giulioz / [MEMEST] Real Professional Project
Loaded

Schematic | Schematic NEW | Code | PCB

SHARE

LIBRARY | OBJECTS | RULES

Designator, part name or selector

Root
Model
Layout

INSPECTOR | CHAT

[MEMEST] Real Professional Project

Description
Arduino Uno shield used to monitor chimney smoke and provide feedback to stove. This shield powers the Arduino using TEGs and a battery. This shield

Created
June 16th 2023
Last updated by giulioz
4 days ago
1 Contributor(s)

Controls

Properties
ADD

Availability & Pricing
Add a Manufacturer Part Number to see inline results.

Updates available for your components DISMISS REVIEW

Stefano J. Attardi
Why We Memo All the Things

- Why We React.memo All Components
- Sane Defaults
- CPU Cost of React.memo
- Memory Cost of React.memo
- Isn’t it Premature Optimization?
- Why we React.useCallback All Callbacks
- Why we React.useMemo All the Props & Deps
- Will Someone Please Think of the Children?
- Conclusion

Stefano J. Attardi

Why We Memo All the Things

October 28, 2020

On my team at Coinbase, we ask everyone to use the React performance trinity – `memo`, `useMemo`, and `useCallback` – all the time. For some reason, this is controversial. I’m guessing this has something to do with Twitter. This article explains why we do it anyway.

Why We React.memo All Components

Let’s start with what we can all agree on: in most apps, some components can benefit from being wrapped in `React.memo`. Maybe because they are expensive to rerender, or maybe they are children of a component that renders much more frequently. Maybe both.

So not using `memo` at all is not an option. We are left with two options:

- Use `memo` some of the time
- Use `memo` all the time

The first option sounds like the most appealing, doesn’t it? Figure out when we can benefit from `React.memo`, and use it then, and only then. However, before we go that far, we have to remind ourselves that we work on a large team. No matter how diligent we are with education, code review, and profiling, **we are not going to get it right 100% of the time**. So we have to ask ourselves:

Why Optimise **Memory Usage**?

Aw, Snap!

Something went wrong while displaying this webpage.

Error code: 5

Learn more

Reload

Out Of Memory Crashes

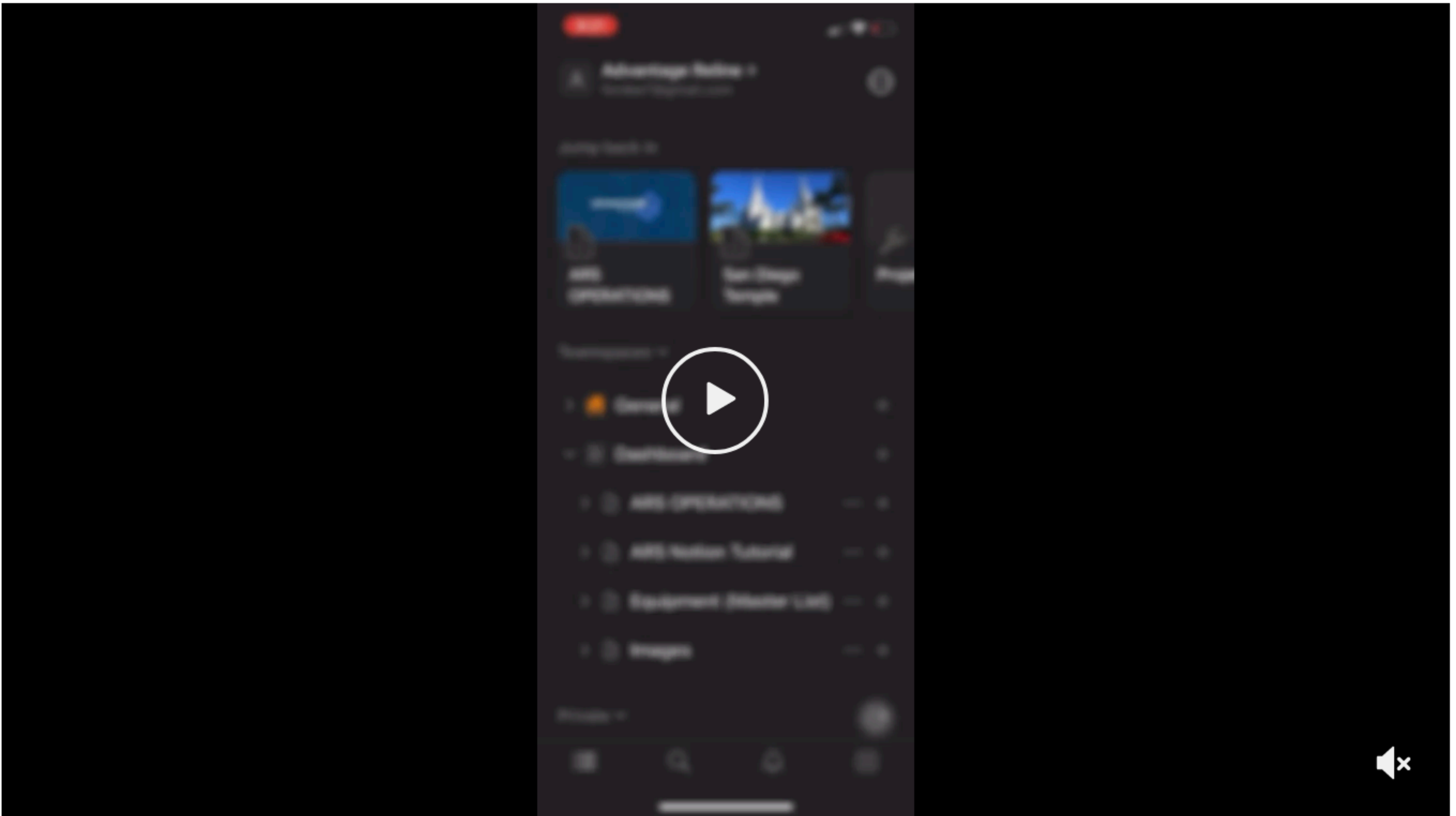
6

Postato da u/fordee7 4 mesi fa

Notion mobile app and browser crash reboot loop - unusable

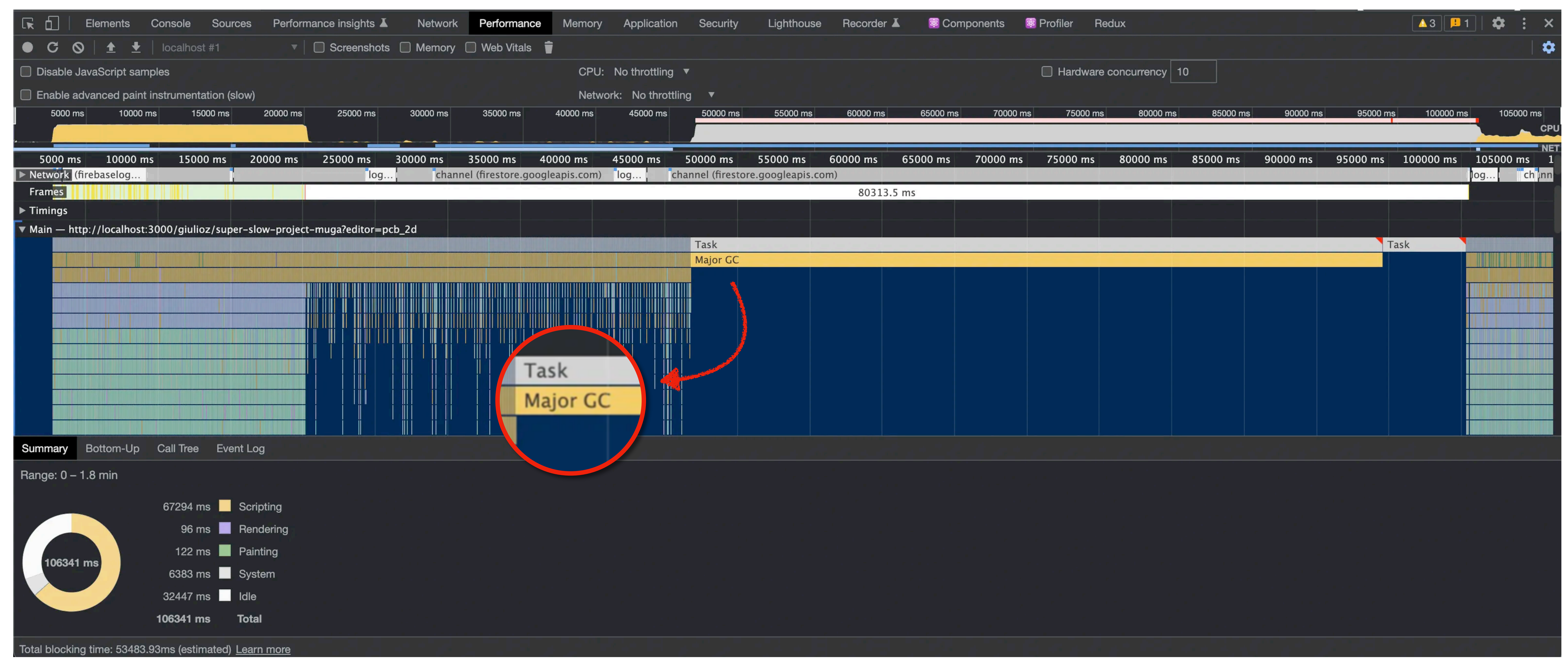
Request/Bug

Notion crashes (goes into a crash/reboot loop) on mobile app and browser with a database of more than about 50-75 rows/pages. We've tested this on multiple phones. All do it - although some work ok when others are crashing. Then tomorrow the users that worked yesterday, crash today. Works good on a computer. This is unacceptable. I've submitted support tickets and so far all i get is a response that they had issues yesterday or the other day but resolved them. I have a team of 6 people having this issue. So bad it is unusable on mobile.



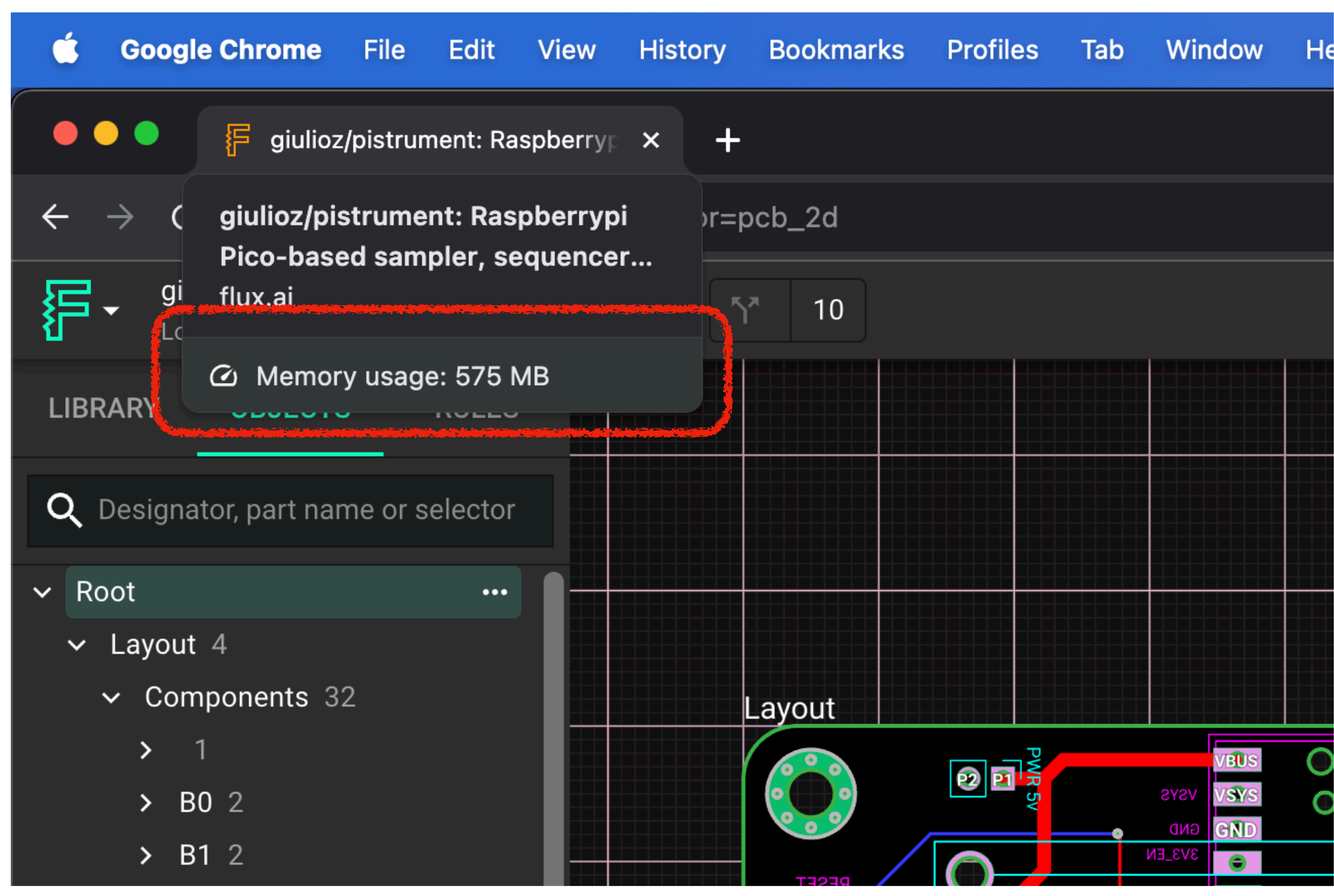
Notion Crash / Reboot Loop

Long **Garbage Collection** times



15300 ms
15400 ms
15500 ms
15600 ms
15700 ms
15800 ms
15900 ms
16000 ms
16100 ms
16200 ms
16300 ms
16400 ms
16500 ms
16600 ms
16700 ms
16800 ms
16900 ms
17000 ms
Task
Animati… Fired
Function Call
loop
(anonymous)
render$1
(an…us)
func2
inv…unc
calc…Viz
Task
Event: pointermove
Function Call
pointerMove
emit
fn
onDrag
onDrag2
drag
snapIfPossible
(anonymous)
(a…s)
(anonymous)
(anonymous)
Major GC
baseExtremum
507.85 ms (self 12 μs) Major GC
Animatio…e Fired
Function Call
loop
(anonymous)
render$1
(ano…ous)
func2
invo…unc
(ano…us)
calc…Viz
(a…)
Task
Eve…ove
han…ove
send…its
intersect
coll…ions
getE…cts
quer…ate
Func…all
sn…le
Task
A…d
Fu…l
loop
(a…)
r…1

Users multitask



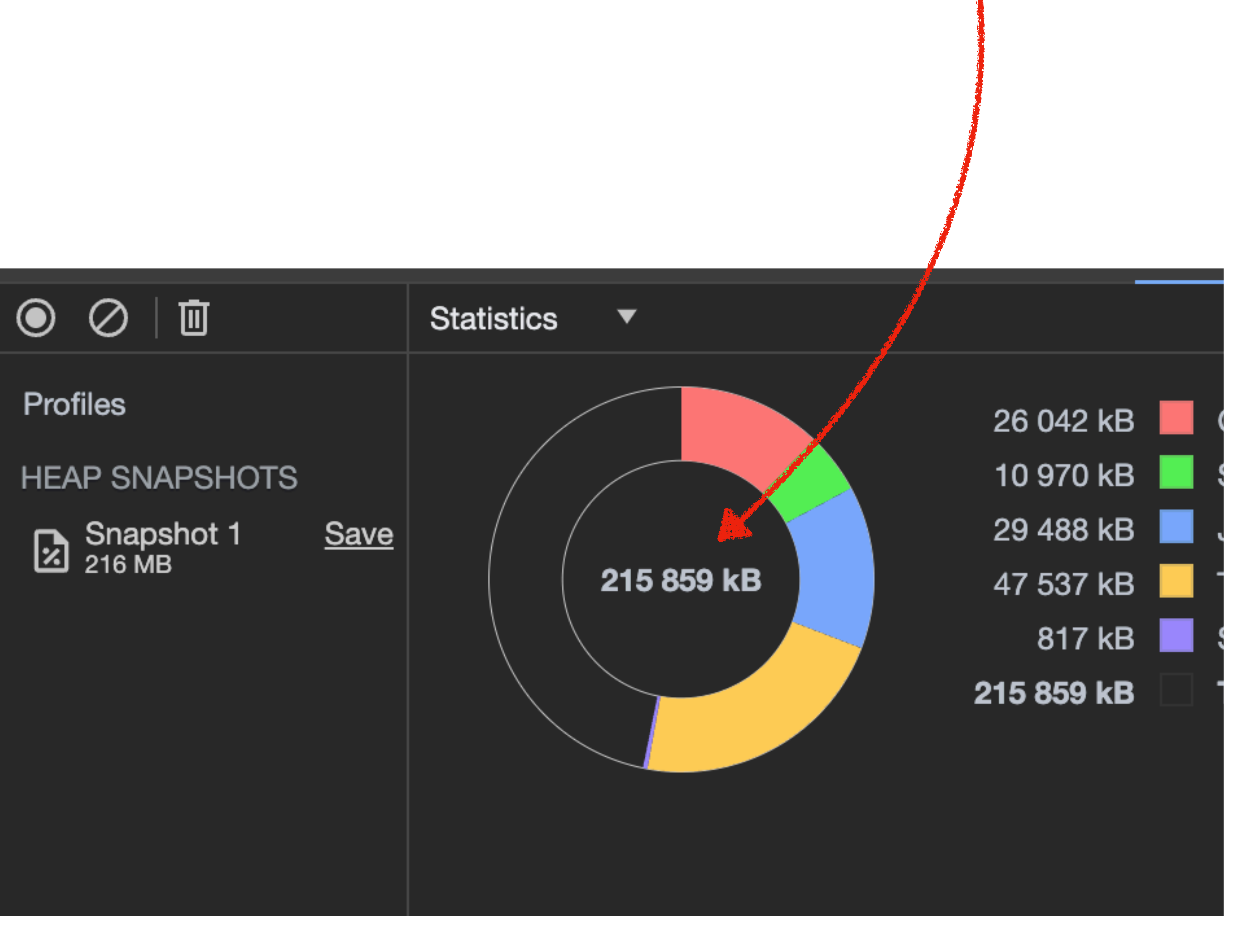
How do we solve this?

1. Identify what occupies memory
2. Kill it with fire 🔥
3. Make sure we don't repeat the same mistake

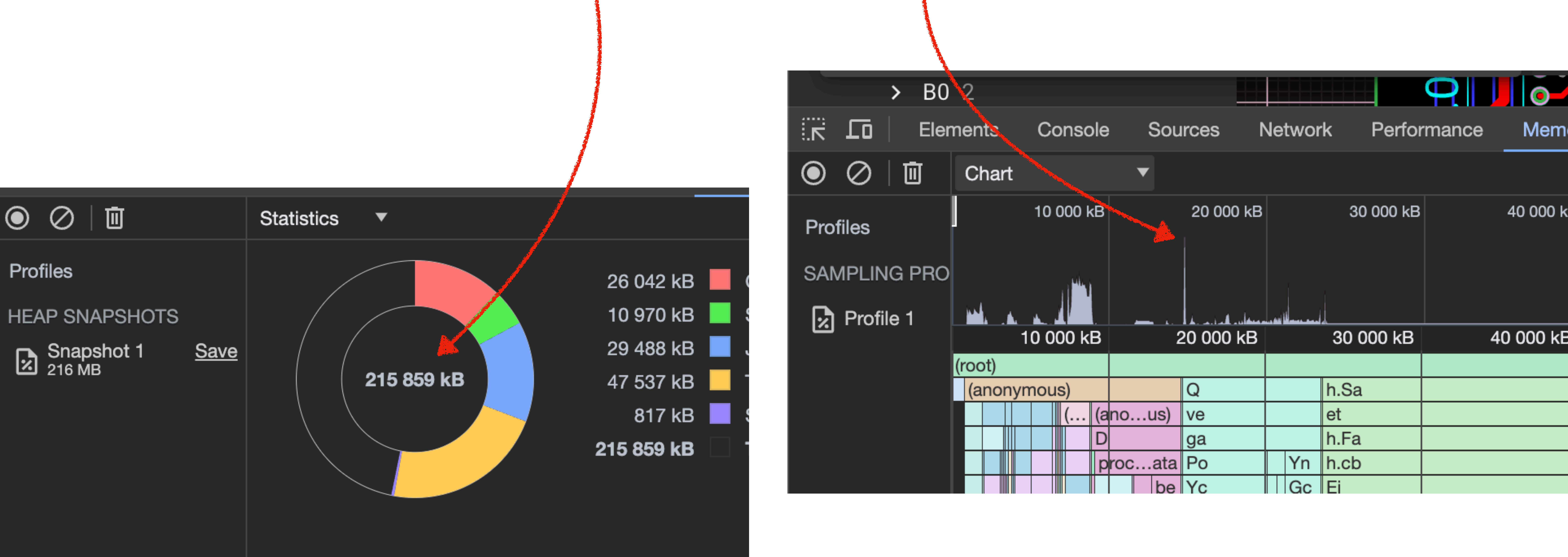
How do we solve this?

1. **Identify what occupies memory**
2. Kill it with fire 🔥
3. Make sure we don't repeat the same mistake

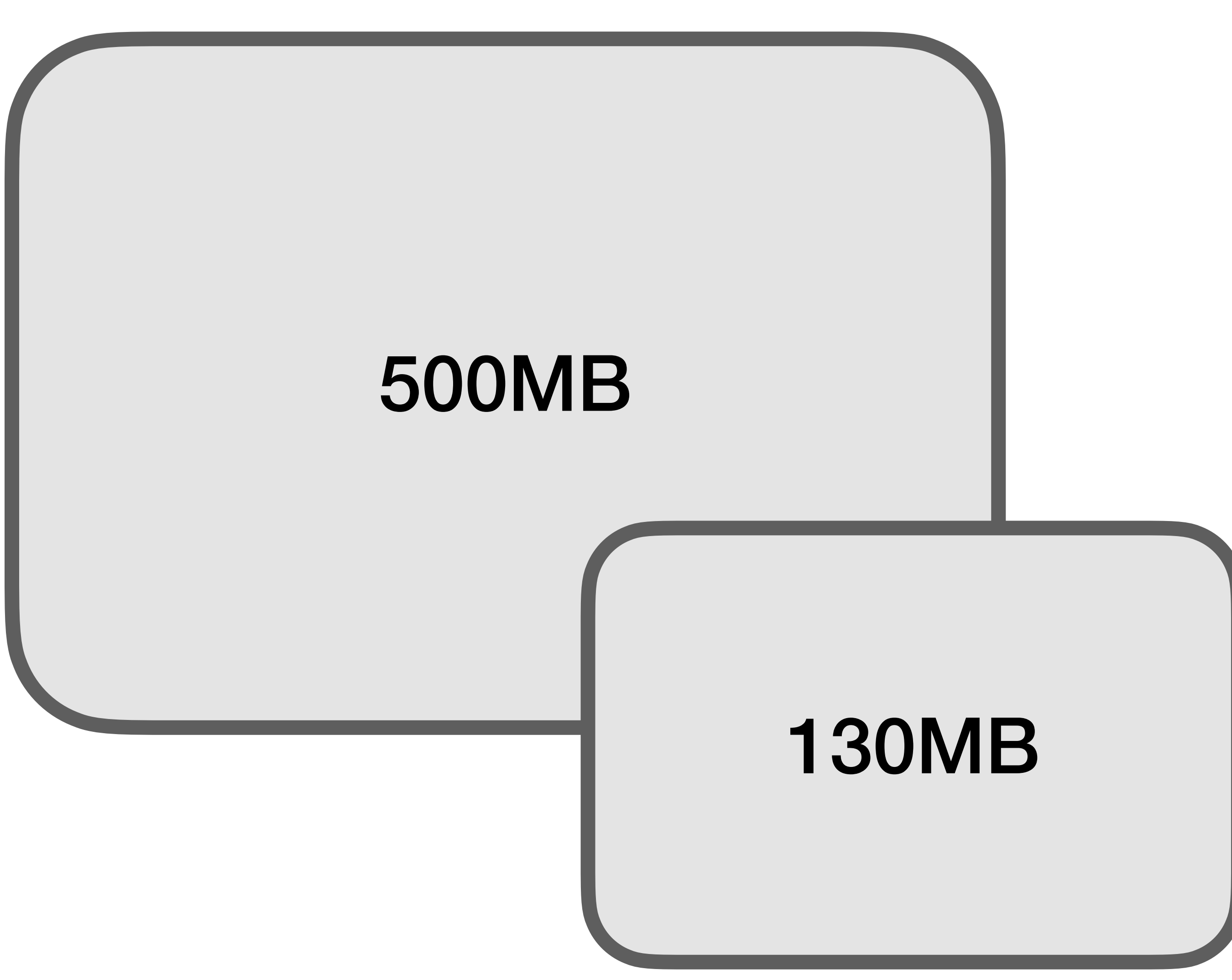
Static or **Transient?**



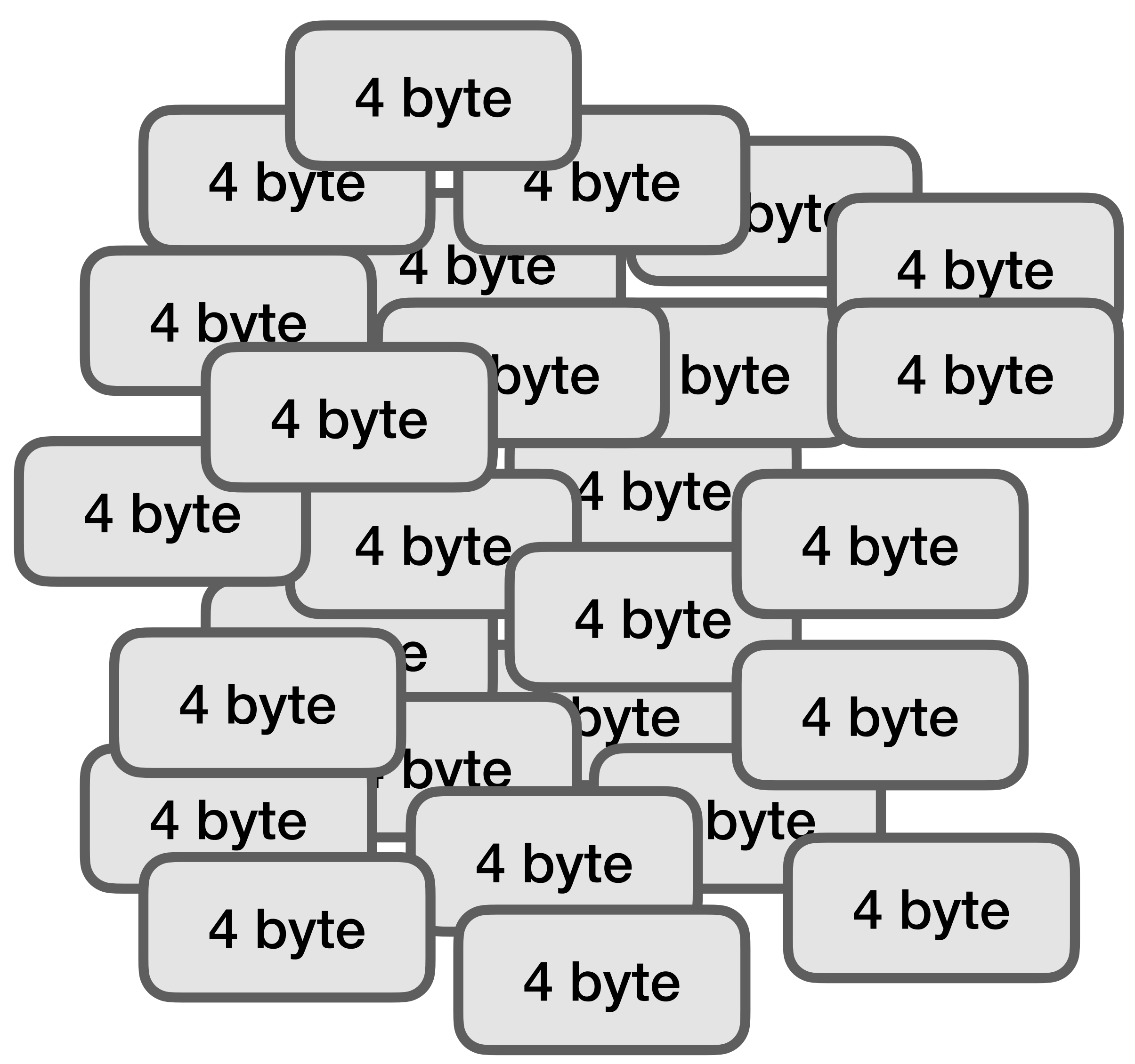
Static or **Transient?**



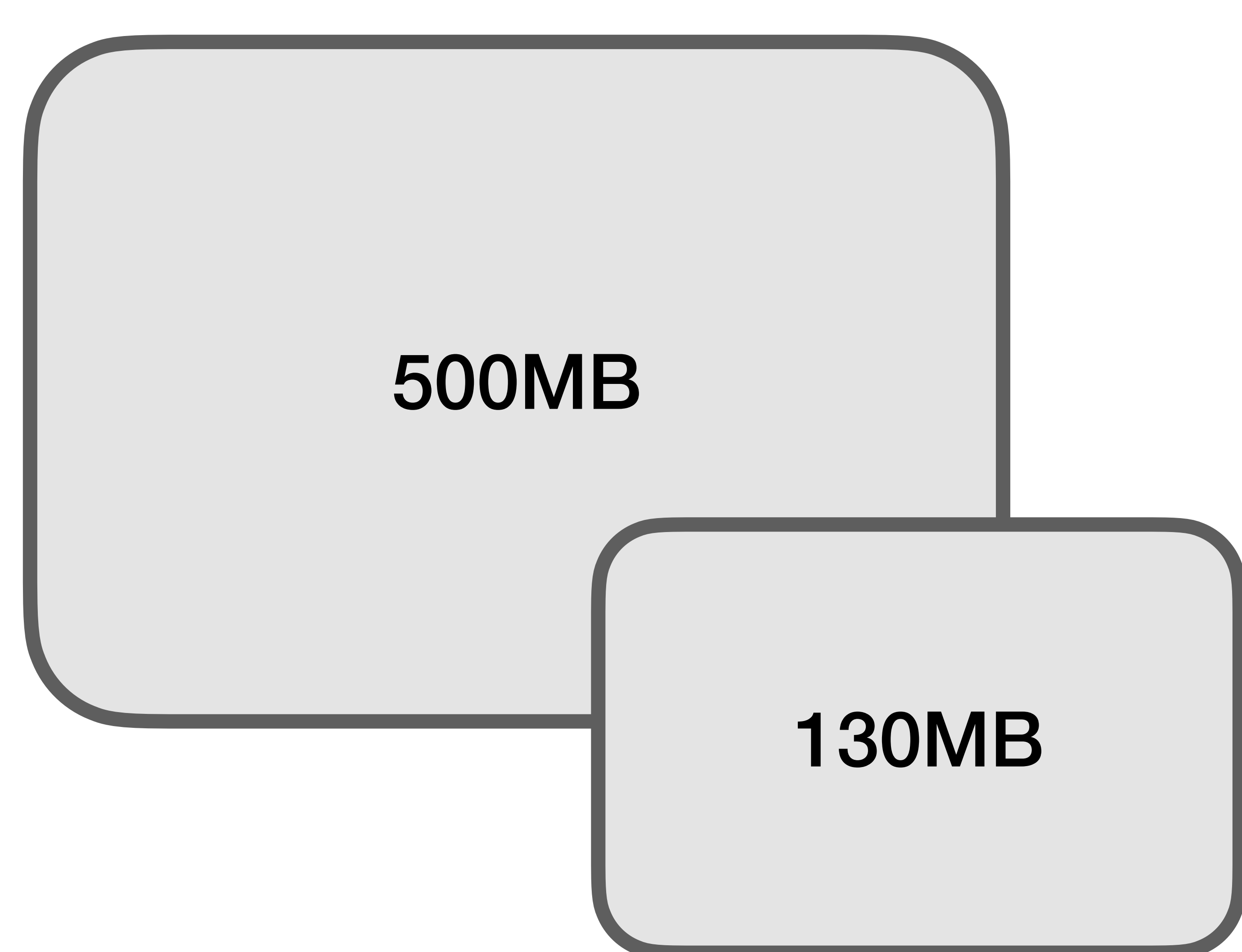
Count vs **Size**



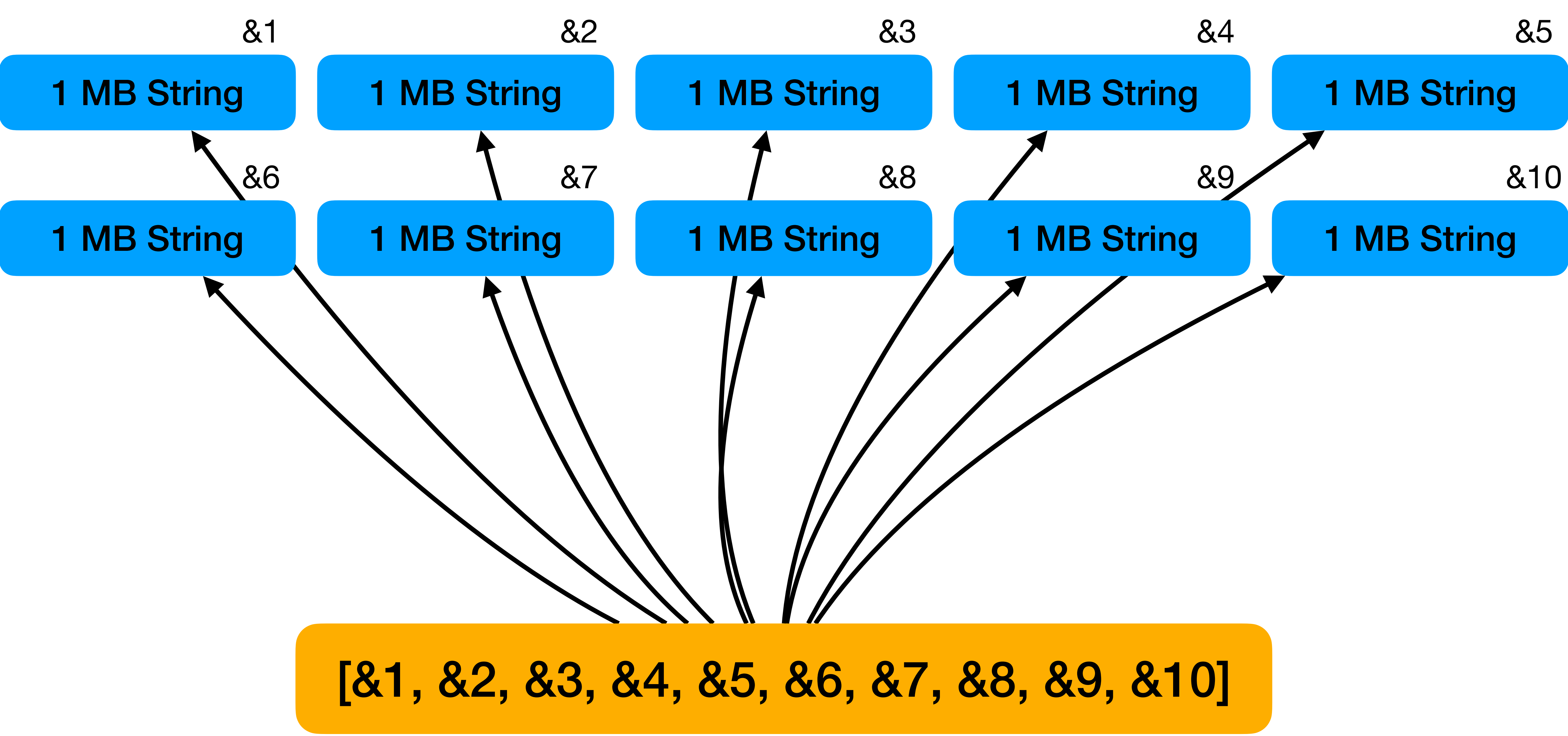
Count vs Size



…can become hundreds of MBs!

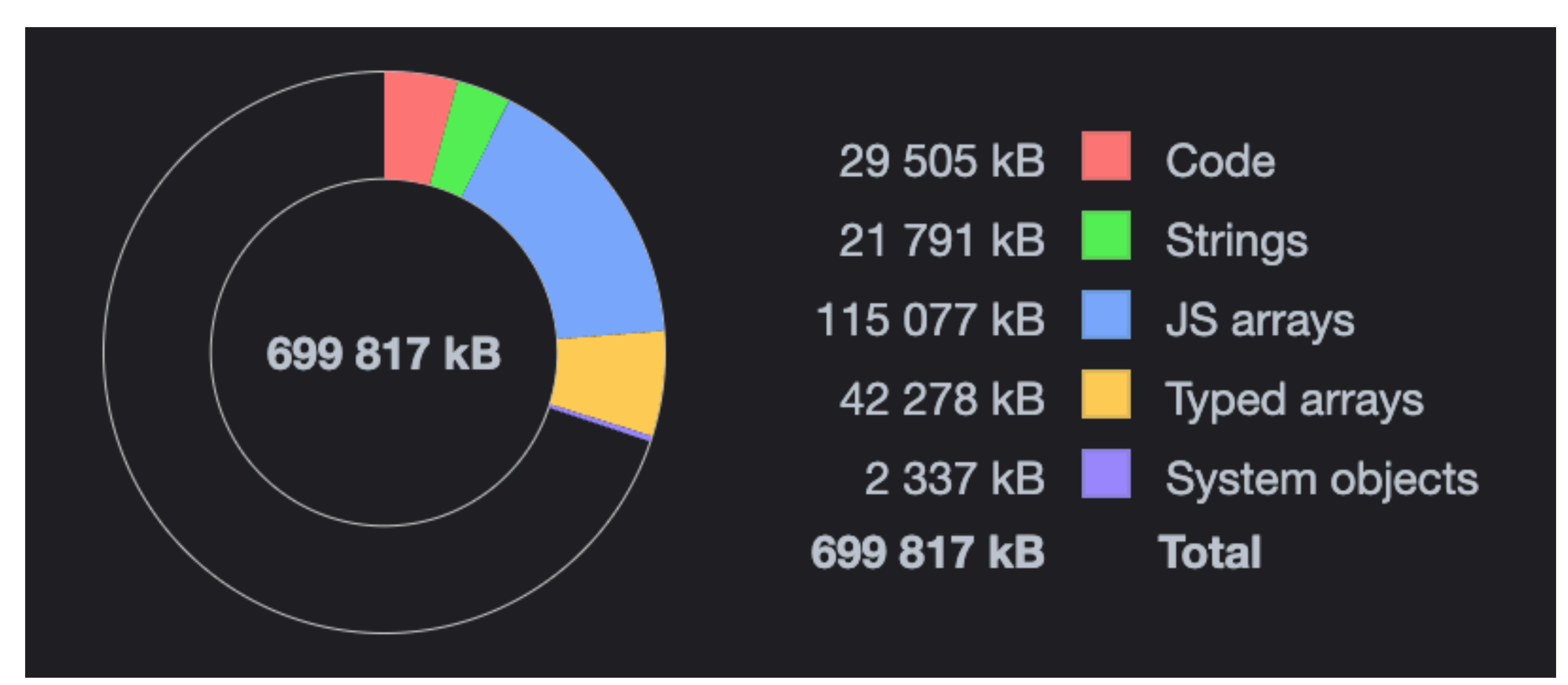


Shallow vs **Retained Size**



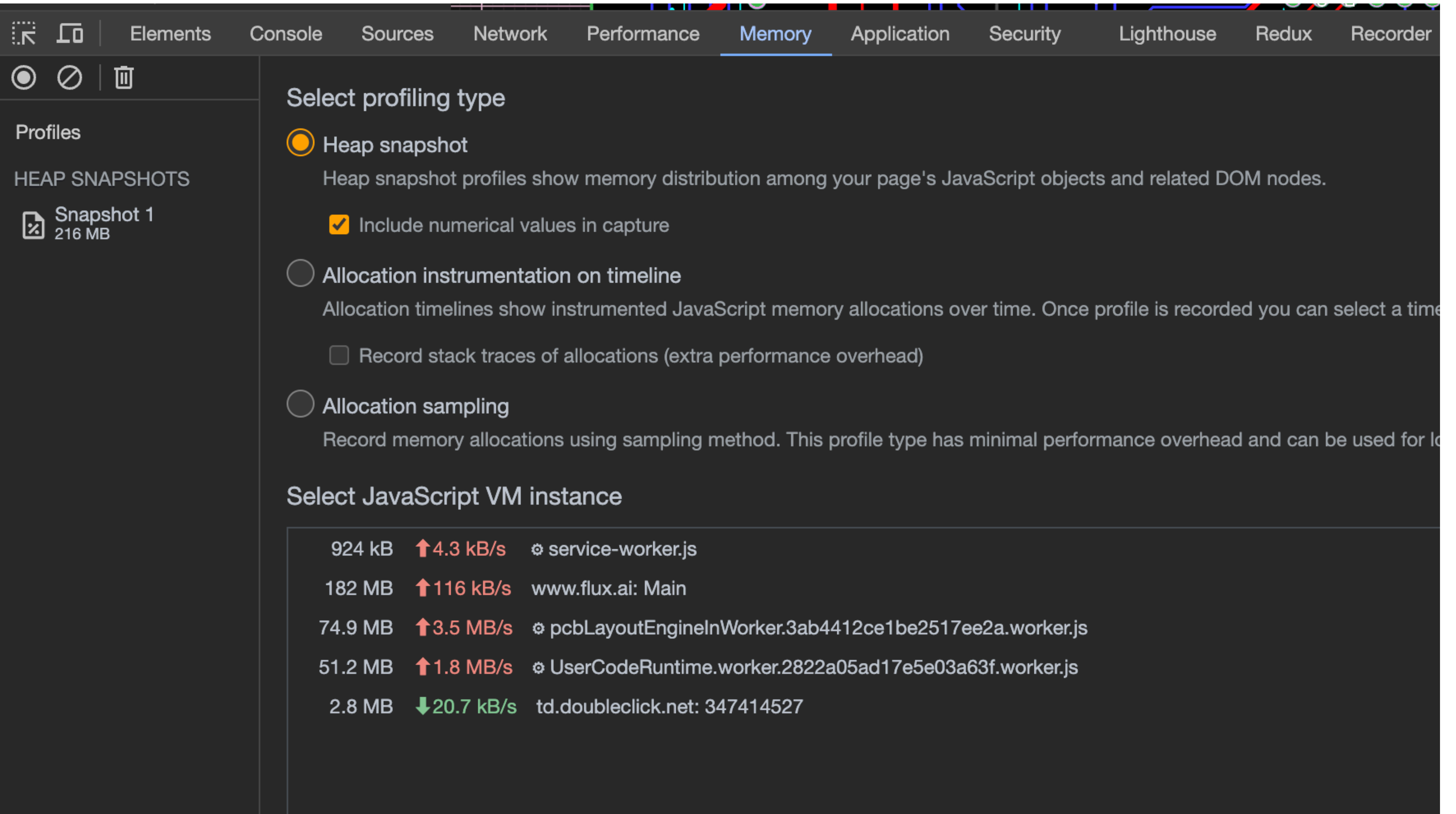
Shallow: 40 bytes
Retained: 40 bytes + 1MB x 10 = **10 MB**

Allocation Types



Tooling 🛠️

Chrome Memory Profiler



Heap **Snapshots**

Elements | Console | Sources | Network | Performance | Memory | Application | Security | Lighthouse | Redux | Recorder

16 | 61 | 69

Summary | Class filter | All objects

Profiles

HEAP SNAPSHOTS

Snapshot 1 — 700 MB — Save

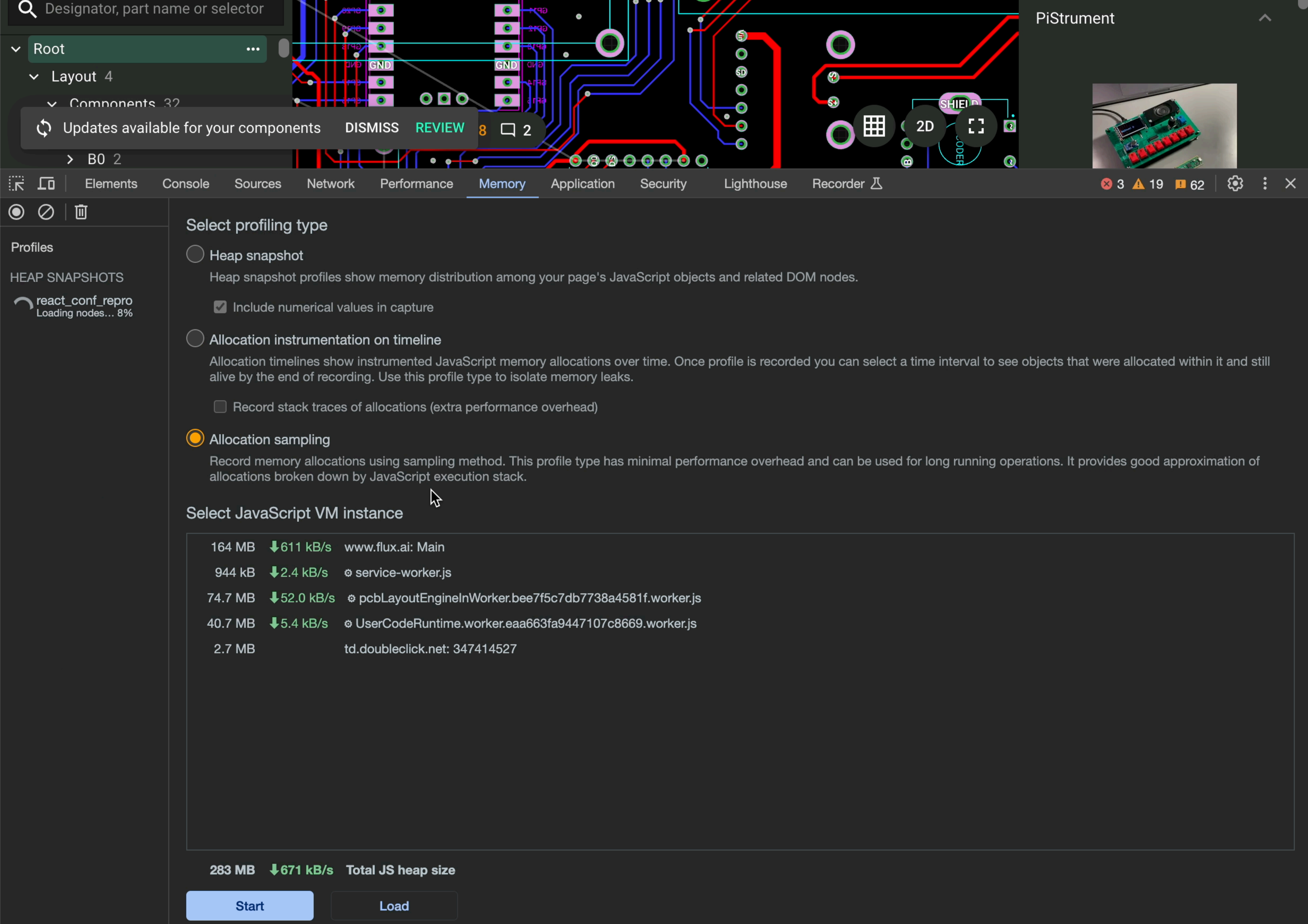
Constructor	Distance	Shallow Size		Retained Size	
Object ×5893405	2	163 272 872	23 %	389 851 294	56 %
Object @1971155	10	12	0 %	17 001 420	2 %
Object @28275291	8	28	0 %	5 258 004	1 %
Object @1454869	8	12	0 %	4 081 560	1 %
Object @35861205	5	12	0 %	3 409 156	0 %
Object @3321121	7	28	0 %	3 281 196	0 %
Object @1445297	7	28	0 %	2 886 556	0 %
Object @2647265	15	28	0 %	2 790 564	0 %
Object @38154489	16	28	0 %	2 766 124	0 %
Object @42704557	10	16	0 %	2 631 588	0 %
Object @42704551	13	12	0 %	2 619 700	0 %
Object @45384767	9	12	0 %	2 301 332	0 %
Object @26217647	10	28	0 %	2 076 424	0 %
Object @24598261	9	12	0 %	1 958 284	0 %
Object @35245061	16	28	0 %	1 577 224	0 %
Object @46340783	16	28	0 %	1 577 224	0 %
Object @46911681	9	28	0 %	1 577 224	0 %
Object @658191	9	68	0 %	1 491 336	0 %

Retainers

Object	Distance	Shallow Size		Retained Size	
wasmModule in system / Context @1971127	9	72	0 %	4 192	0 %
context in ClipperLibWrapper2() @1971157 index.ts:75	8	32	0 %	260	0 %
ClipperLibWrapper in Object @3202445	7	12	0 %	3 248	0 %
exports in Object @3202443	6	24	0 %	788	0 %
../node_modules/js-angusj-clipper/universal/index.js in Object @35861205	5	12	0 %	3 409 156	0 %
__webpack_module_cache__ in system / Context @369805	4	28	0 %	28	0 %
previous in system / Context @35860783	3	52	0 %	404	0 %

Allocation Sampling

Let's play with the memory profiler



Designator, part name or selector
Root
Layout 4
Components 32
Updates available for your components
DISMISS
REVIEW
B0 2
PiStrument
2D
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
Profiles
HEAP SNAPSHOTS
react_conf_repro
Loading nodes… 8%
Select profiling type
Heap snapshot
Heap snapshot profiles show memory distribution among your page's JavaScript objects and related DOM nodes.
Include numerical values in capture
Allocation instrumentation on timeline
Allocation timelines show instrumented JavaScript memory allocations over time. Once profile is recorded you can select a time interval to see objects that were allocated within it and still alive by the end of recording. Use this profile type to isolate memory leaks.
Record stack traces of allocations (extra performance overhead)
Allocation sampling
Record memory allocations using sampling method. This profile type has minimal performance overhead and can be used for long running operations. It provides good approximation of allocations broken down by JavaScript execution stack.
Select JavaScript VM instance
164 MB 611 kB/s www.flux.ai: Main
944 kB 2.4 kB/s service-worker.js
74.7 MB 52.0 kB/s pcbLayoutEngineInWorker.bee7f5c7db7738a4581f.worker.js
40.7 MB 5.4 kB/s UserCodeRuntime.worker.eaa663fa9447107c8669.worker.js
2.7 MB td.doubleclick.net: 347414527
283 MB 671 kB/s Total JS heap size
Start
Load

```
14  // We keep all the useFrames in a map with uuids, so we can keep track of all of them removing the old ones
15  const fastUseFrameEvents = new Map<string, FrameCallback>();
16
17  export function useFrameFast(fn: FrameCallback) {
18      // UUID created only the first time
19      const [myId] = useState(() => uuid());
20
21      // On unmount we clear the record
22      useEffect(() => () => void fastUseFrameEvents.delete(myId), [myId]);
23
24      // At every render we have a new callback, so we update it (potentially dangerous with concurrent mode?)
25      fastUseFrameEvents.set(myId, fn);
26  }
27
```

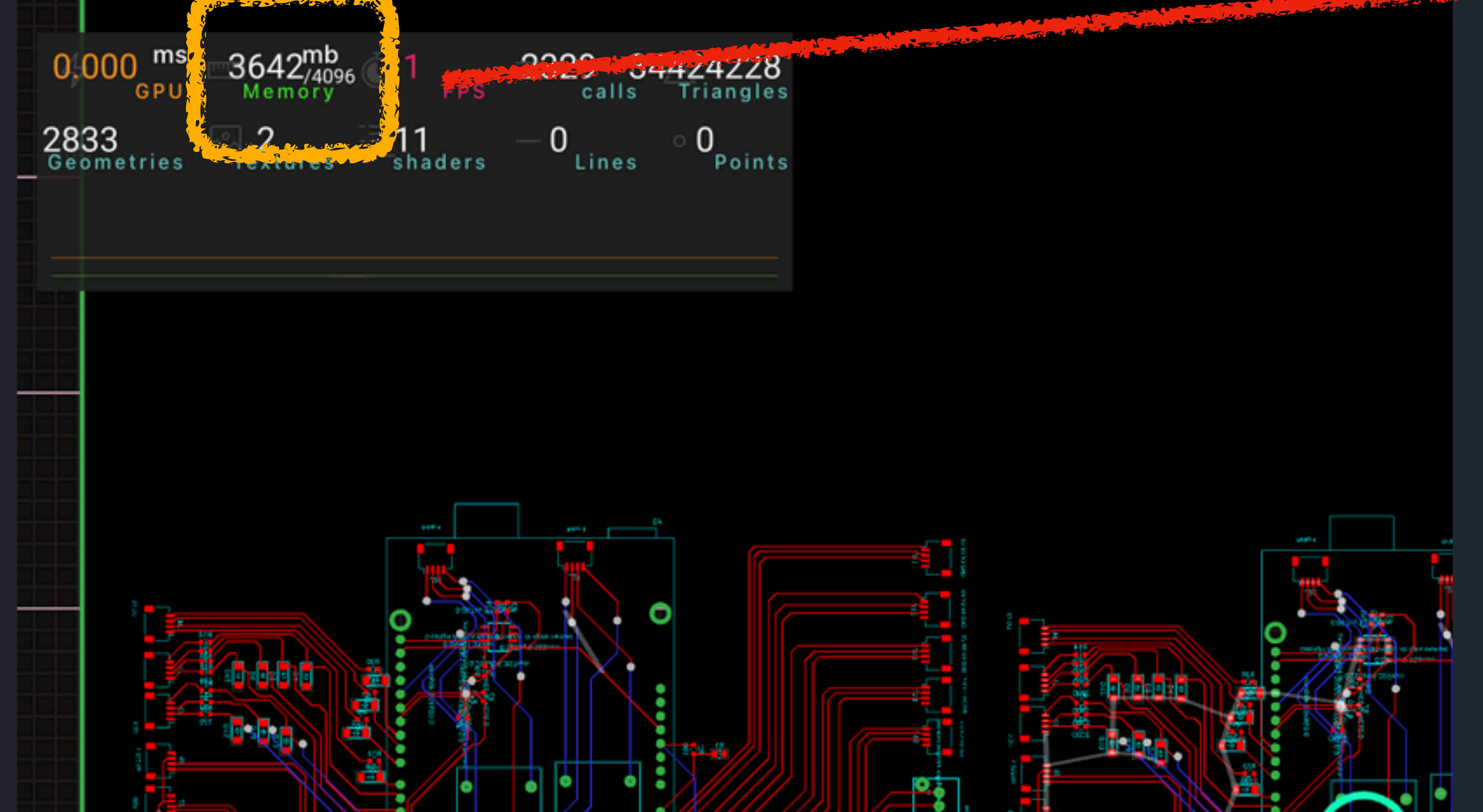
🚨 ***called thousands of times!***

```diff
13
14 - // We keep all the useFrames in a map with uuids, so we can keep track of all of
     them removing the old ones
15 - // (exported only for testing purposes)
16 - export const fastUseFrameEvents = new Map<string, FrameCallback>();
17
18   export function useFrameFast(fn: FrameCallback) {
19 -     // UUID created only the first time
20 -     const [myId] = useState(() => uuid());
21 -
22 -     // On unmount we clear the record
23 -     useEffect(() => () => void fastUseFrameEvents.delete(myId), [myId]);
24
```

```diff
12
13 + export const fastUseFrameEvents = new Set<FrameCallback>();
14
15   export function useFrameFast(fn: FrameCallback) {
16 +     useLayoutEffect(() => {
17 +         fastUseFrameEvents.add(fn);
18 +         return () => void fastUseFrameEvents.delete(fn);
19 +     }, [fn]);
20 + }
21
```

On a 375 parts document there is almost a 50% improvement.

Before (the browser almost crashed because of an OOM):

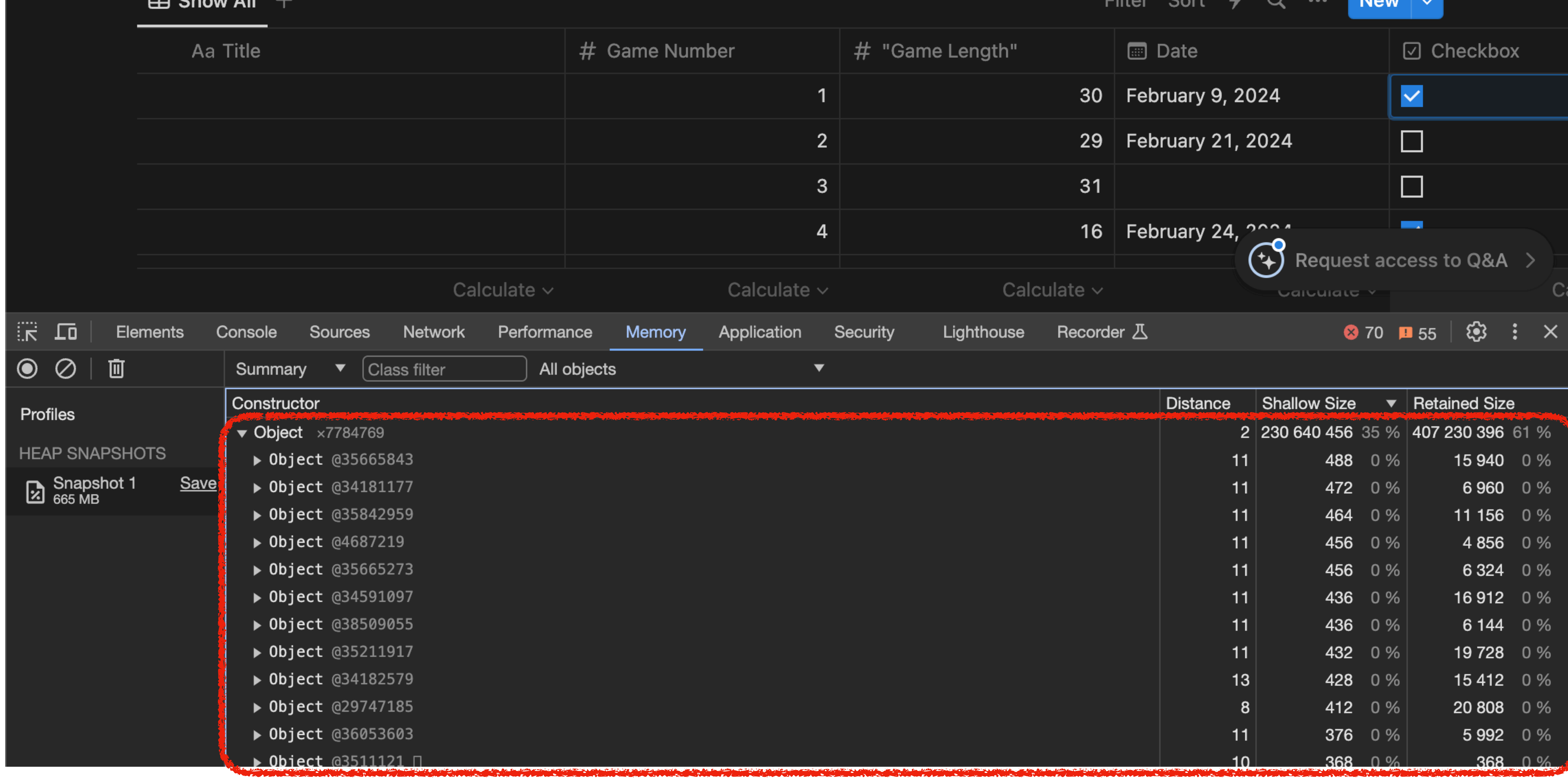


After:

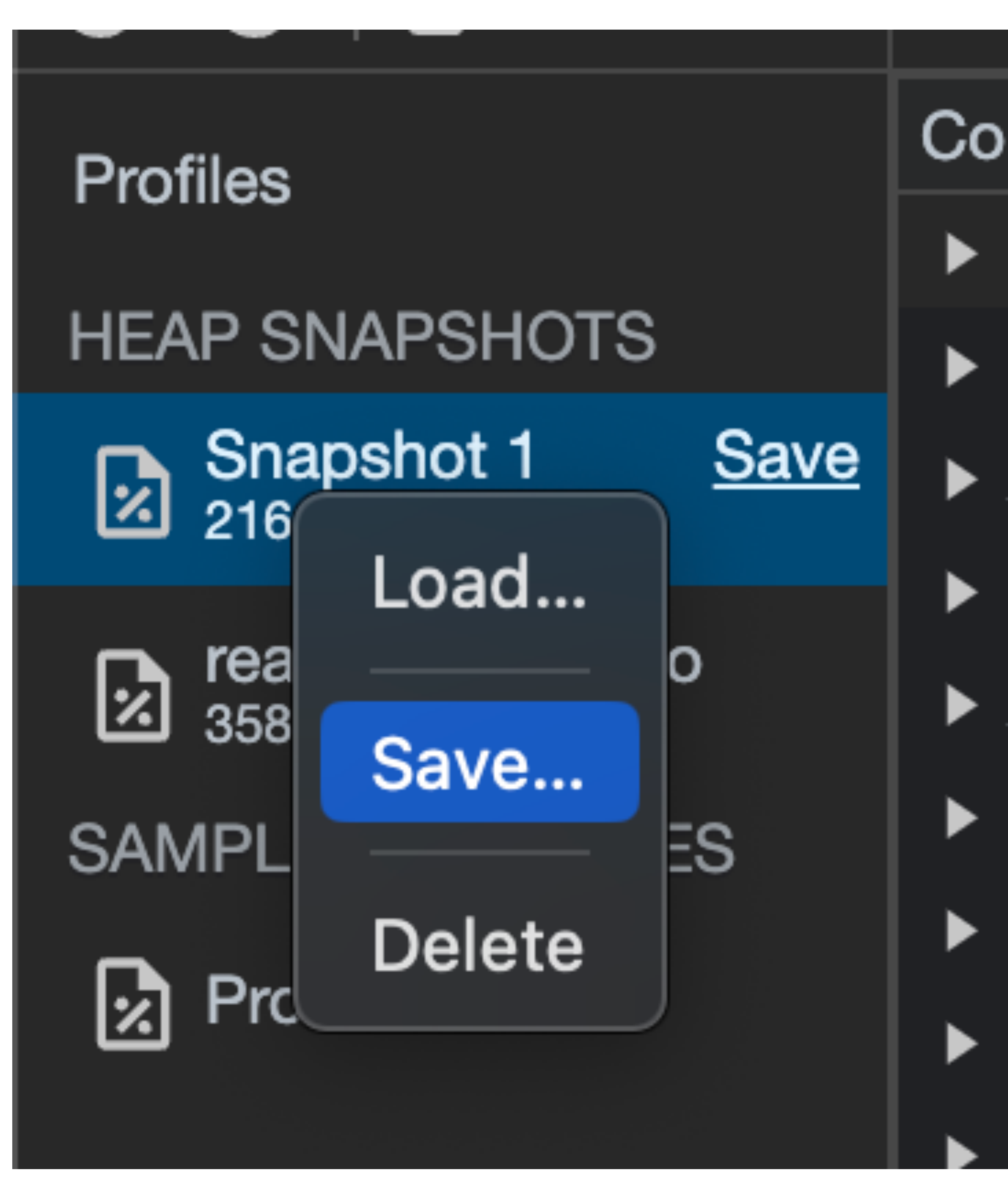
*Those are simply **too many!***

Constructor	Distance	Shallow Size	Retained Size
▼ Object ×2254628	2	62 534 152 17 %	199 263 952 56 %
▶ Object @6415517	3	12 0 %	[illegible] 5 %
▶ Object @5447681	4	12 0 %	10 868 616 3 %
▶ Object @3086099	6	28 0 %	5 190 988 1 %
▶ Object @3083509	6	12 0 %	3 277 032 1 %
▶ Object @2908085	6	12 0 %	2 174 664 1 %
▶ Object @4066881	9	164 0 %	1 282 884 0 %
▶ Object @4066883	9	12 0 %	1 280 500 0 %
▶ Object @4842229	8	36 0 %	1 225 608 0 %
▶ Object @12782069	9	28 0 %	1 224 440 0 %
▶ Object @4067157	7	196 0 %	1 221 428 0 %
▶ Object @5338541	7	16 0 %	1 190 544 0 %
▶ Object @5345999	10	12 0 %	1 181 120 0 %
▶ Object @1179777	19	28 0 %	1 095 972 0 %
▶ Object @10423241	10	12 0 %	1 088 576 0 %
▶ Object @3150795	6	12 0 %	935 548 0 %
▶ Object @5342799	7	40 0 %	888 980 0 %
▶ Object @5340719	10	12 0 %	882 620 0 %
▶ Object @7330603 ▯	5	24 0 %	866 176 0 %
▶ Object @6105837 ▯	6	12 0 %	866 132 0 %
▶ Object @4066887	9	12 0 %	653 052 0 %
▶ Object @10196345	7	20 0 %	598 324 0 %
▶ Object @10578239	10	12 0 %	586 296 0 %

Show All
Filter
Sort
New
Aa Title
Game Number
"Game Length"
Date
Checkbox
1
30
February 9, 2024
2
29
February 21, 2024
3
31
4
16
February 24, 2024
Request acce
Calculate
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
70
Summary
Class filter
All objects
Profiles
HEAP SNAPSHOTS
Snapshot 1
Saving… 66%
Save
Constructor
Distance
Shallow Size
Object ×7784769
2
230 640 456 35 %
(array) ×3708278
2
104 772 644 16 %
(closure) ×3132543
2
88 876 384 13 %
Array ×4135592
2
66 170 124 10 %
system / Context ×1726054
3
53 462 116 8 %
(compiled code) ×350869
3
24 273 380 4 %
(string) ×447266
2
19 756 872 3 %
Rl ×173866
5
17 386 600 3 %
(system) ×432067
2
13 182 772 2 %
(concatenated string) ×339632
3
6 792 640 1 %
g ×98419
8
5 518 108 1 %



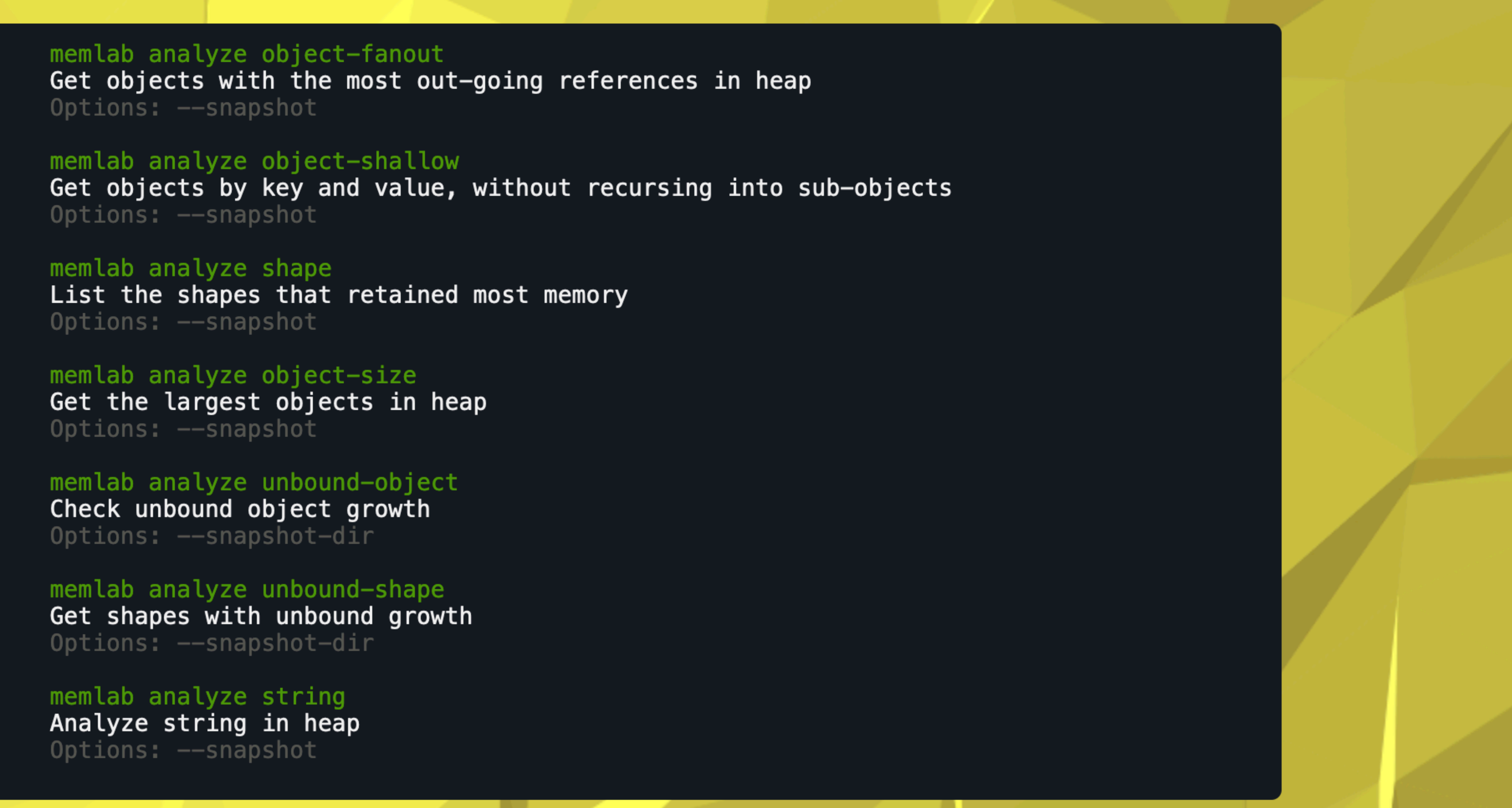
snakes_count_1000
Show All
Filter
Sort
New
Aa Title
Game Number
"Game Length"
Date
Checkbox
1
30
February 9, 2024
2
29
February 21, 2024
3
31
4
16
Request access to Q&A
Calculate
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
70
55
Summary
Class filter
All objects
Profiles
HEAP SNAPSHOTS
Snapshot 1
665 MB
Save
Constructor
Distance
Shallow Size
Retained Size
Object ×7784769
2
230 640 456 35 %
407 230 396 61 %
Object @35665843
Object @34181177
Object @35842959
Object @4687219
Object @35665273
Object @34591097
Object @38509055
Object @35211917
Object @34182579
Object @29747185
Object @36053603


Profiles
HEAP SNAPSHOTS
Snapshot 1
Save
Load...
Save...
Delete
Co

memlab

Analyzes JavaScript heap and finds memory leaks in browser and node.js

Learn more



Define Your Test

Define E2E test scenarios on browser interaction:

```
// test.js
function url() {
  return 'https://www.google.com/maps/place/Sili
}
async function action(page) {
  await page.click('button[aria-label="Hotels"]'
}
async function back(page) {
  await page.click('[aria-label="Close"]');
}

module.exports = {action, back, url};
```

Run memlab in CLI

Find memory leaks with the custom E2E test scenario:

```
$ memlab run --scenario test.js
```

Support memory analyses for the previous browser test:

```
# Analyze duplicated string in heap
$ memlab analyze string
# Check unbound object growth
$ memlab analyze unbound-object
# Get shapes with unbound growth
$ memlab analyze unbound-shape
# Discover more memory analyses
$ memlab analyze -h
```

Programming API

Memory analysis for JavaScript heap snapshots:

```
const {findLeaks, takeSnapshots} = require('@mem

async function test() {
  const scenario = {
    url: () => 'https://www.facebook.com',
  };
  const result = await takeSnapshots({scenario})
  const leaks = findLeaks(result);
  // ...
}
```

Powerful **API** for **snapshots**!

```
16
17  class MyAnalysisTest extends BaseAnalysis {
18      override getCommandName(): string {
19          return "my_analysis";
20      }
21
22      override getDescription(): string {
23          return "Example Analysis";
24      }
25
26      override async process(options: HeapAnalysisOptions): Promise<void> {
27          const snapshotPath = pluginUtils.getSnapshotFileForAnalysis(options);
28          const snapshot = await utils.getSnapshotFromFile(snapshotPath, {buildNodeIdIndex: true, verbose: true});
29          analysis.preparePathFinder(snapshot);
30
31          // Do stuff here with the snapshot!
32      }
33  }
34
35  export default MyAnalysisTest;
36
```

Which **types of objects** are taking up the most space, out of the 2 millions we found in the snapshot?

Constructor	Distance	Shallow Size		Retained Size ▼	
▼ Object ×2254628	2	62 534 152	17 %	199 263 952	56 %
▶ Object @6415517	9	12	0 %	16 939 416	5 %
▶ Object @5447681	4	12	0 %	10 868 616	3 %
▶ Object @3086099	6	28	0 %	5 190 988	1 %
▶ Object @3083509	6	12	0 %	3 277 032	1 %
▶ Object @2908085	6	12	0 %	2 174 664	1 %
▶ Object @4066881	9	164	0 %	1 282 884	0 %
▶ Object @4066883	9	12	0 %	1 280 500	0 %
▶ Object @4842229	8	36	0 %	1 225 608	0 %
▶ Object @12782069	9	28	0 %	1 224 440	0 %
▶ Object @4067157	7	196	0 %	1 221 428	0 %
▶ Object @5338541	7	16	0 %	1 190 544	0 %
▶ Object @5345999	10	12	0 %	1 181 120	0 %
▶ Object @1179777	19	28	0 %	1 095 972	0 %
▶ Object @10423241	10	12	0 %	1 088 576	0 %
▶ Object @3150795	6	12	0 %	935 548	0 %
▶ Object @5342799	7	40	0 %	888 980	0 %

```typescript
override async process(options: HeapAnalysisOptions): Promise<void> {
    const snapshotPath = pluginUtils.getSnapshotFileForAnalysis(options);
    const snapshot = await utils.getSnapshotFromFile(snapshotPath, {buildNodeIdIndex: true, verbose: true});
    analysis.preparePathFinder(snapshot);

    info.overwrite("Breaking down memory by shapes...");
    const breakdown: Record<string, HeapNodeIdSet> = Object.create(null);
    const population: Record<string, {examples: IHeapNode[]; n: number}> = Object.create(null);

    // group objects based on their shapes
    snapshot.nodes.forEach((node: IHeapNode) => {
        if ((node.type !== "object" && !utils.isStringNode(node)) || config.nodeIgnoreSetInShape.has(node.name)) {
            return;
        }
        const key = serializer.summarizeNodeShape(node);
        breakdown[key] = breakdown[key] || new Set();
        breakdown[key].add(node.id);
        if (population[key] === undefined) {
            population[key] = {examples: [], n: 0};
        }
        ++population[key].n;
        // retain the top 5 examples
        const examples = population[key].examples;
        examples.push(node);
        examples.sort((n1, n2) => n2.self_size - n1.self_size);
        if (examples.length > 5) {
            examples.pop();
        }
    });

    // calculate and sort based on shallow sizes
    const ret: Array<{key: string; size: number}> = [];
    let sum = 0;
    for (const key in breakdown) {
        let size = 0;
        breakdown[key].forEach((nodeId) => {
            const node = snapshot.getNodeById(nodeId);
            size += node?.self_size ?? 0;
            sum += node?.self_size ?? 0;
        });
        ret.push({key, size});
    }
    ret.sort((o1, o2) => o2.size - o1.size);

    info.topLevel("Object shapes with top shallow sizes:");
    info.lowLevel(" (Use `memlab trace --node-id=@ID` to get trace)\n");
    const topList = ret.slice(0, 40);
    let topListSum = 0;
    const opt = {color: true, compact: true};
    const colon = chalk.grey(": ");

    // print the shapes with the biggest shallow size
    for (const o of topList) {
        const {examples} = population[o.key];
        const shapeStr = serializer.summarizeNodeShape(examples[0], opt);
        const bytes = utils.getReadableBytes(o.size);
        info.topLevel(`${shapeStr}${colon}${bytes} (${utils.getReadablePercent(o.size / sum)})`);
        topListSum += o.size;
    }

    info.topLevel(`Different object types: ${ret.length}`);
    info.topLevel(`Remaining object types: ${utils.getReadableBytes(sum - topListSum)}`);
    info.topLevel(`Total object memory usage: ${utils.getReadableBytes(sum)}`);
}
```

1. Load the Snapshot

2. Find all the object types

3. Compute total shallow size for each type

4. Sort and print results

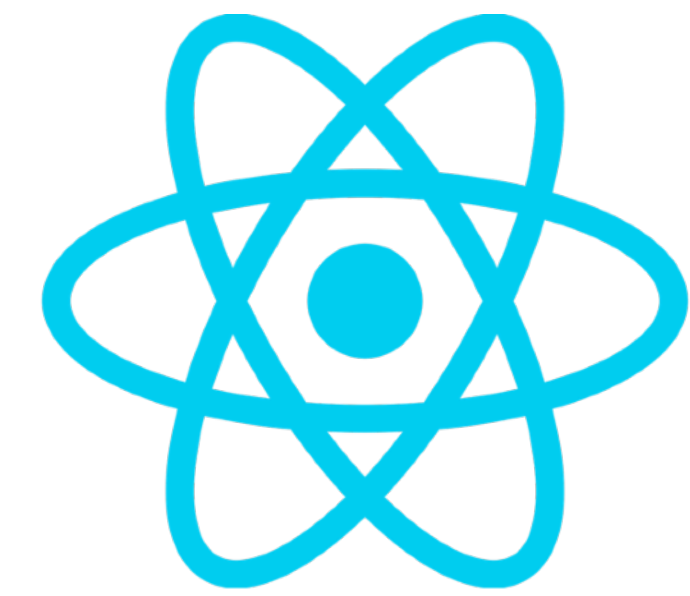
```
52  Object { baseQueue, baseState, memoizedState, next, queue }: 52.2MB (24.73%)
53  Array: 24.9MB (11.8%)
54  FiberNode: 23.3MB (11.05%)
55  string: 15.3MB (7.26%)
56  Object { current }: 13.5MB (6.43%)
57  Object { create, deps, destroy, next }: 13MB (6.16%)
58  Vector3: 5MB (2.39%)
59  Object {  }: 4.5MB (2.15%)
60  Object { dispatch, interleaved, lastRenderedReducer, pending }: 4.2MB (2.02%)
61  Object { baseQueue, next, queue }: 4.2MB (2.02%)
62  Vector2: 3.7MB (1.79%)
63  Group: 2.6MB (1.26%)
64  Matrix4: 2.5MB (1.2%)
65  Object { context, memoizedValue, next }: 2.5MB (1.19%)
66  Euler: 1.9MB (0.93%)
67  Quaternion: 1.8MB (0.89%)
68  Object { handlers, memoizedProps, previousAttach, root, type, ... }: 1.7MB (0.82%)
69  LineCurve: 1.7MB (0.81%)
70  HyperInstancingPlaceholder: 1.6MB (0.8%)
71  BufferGeometry: 1.4MB (0.67%)
72  Object { dispatch, interleaved, lastRenderedReducer, lastRenderedState, pending }: 1.3MB (0.65%)
73  Object { lastEffect, stores }: 1.3MB (0.65%)
74  Object { $$typeof, key, props, ref, type, ... }: 1.2MB (0.6%)
75  Object { firstContext }: 1.1MB (0.55%)
76  Mesh: 1MB (0.51%)
77  CirclePlaceholder: 1MB (0.48%)
```

```
52  Object { baseQueue, baseState, memoizedState, next, queue }: 52.2MB (24.73%)
53  Array: 24.9MB (11.8%)
54  FiberNode: 23.3MB (11.05%)
55  string: 15.3MB (7.26%)
56  Object { current }: 13.5MB (6.43%)
57  Object { create, deps, destroy, next }: 13MB (6.16%)
58  Vector3: 5MB (2.39%)
59  Object {  }: 4.5MB (2.15%)
60  Object { dispatch, interleaved, lastRenderedReducer, pending }: 4.2MB (2.02%)
61  Object { baseQueue, next, queue }: 4.2MB (2.02%)
62  Vector2: 3.7MB (1.79%)
63  Group: 2.6MB (1.26%)
64  Matrix4: 2.5MB (1.2%)
65  Object { context, memoizedValue, next }: 2.5MB (1.19%)
66  Euler: 1.9MB (0.93%)
67  Quaternion: 1.8MB (0.89%)
68  Object { handlers, memoizedProps, previousAttach, root, type, ... }: 1.7MB (0.82%)
69  LineCurve: 1.7MB (0.81%)
70  HyperInstancingPlaceholder: 1.6MB (0.8%)
71  BufferGeometry: 1.4MB (0.67%)
72  Object { dispatch, interleaved, lastRenderedReducer, lastRenderedState, pending }: 1.3MB (0.65%)
73  Object { lastEffect, stores }: 1.3MB (0.65%)
74  Object { $$typeof, key, props, ref, type, ... }: 1.2MB (0.6%)
75  Object { firstContext }: 1.1MB (0.55%)
76  Mesh: 1MB (0.51%)
77  CirclePlaceholder: 1MB (0.48%)
```

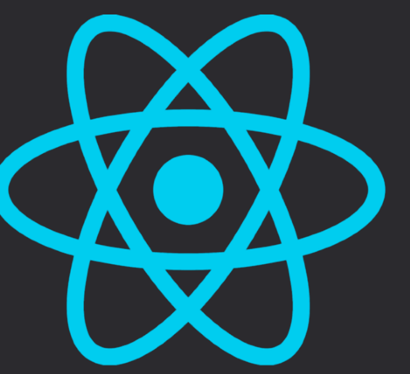
react / packages / react-reconciler / src / ReactFiberHooks.js

Code | Blame | 4853 lines (4519 loc) · 156 KB

```
183
184  export type Hook = {
185    memoizedState: any,
186    baseState: any,
187    baseQueue: Update<any, any> | null,
188    queue: any,
189    next: Hook | null,
190  };
191
```



Keeping track of hooks is expensive!



```
52  Object { baseQueue, baseState, memoizedState, next, queue }: 52.2MB (24.73%)
53  Array: [illegible]
54  FiberNode: 23.3MB (11.05%)
55  string: 15.3MB (7.26%)
56  Object { current }: 13.5MB (6.43%)
57  Object { create, deps, destroy, next }: 13MB (6.16%)
58  Vector3: 5MB (2.39%)
59  Object {  }: 4.5MB (2.15%)
60  Object { dispatch, interleaved, lastRenderedReducer, pending }: 4.2MB (2.02%)
61  Object { baseQueue, next, queue }: 4.2MB (2.02%)
62  Vector2: 3.7MB (1.79%)
63  Group: 2.6MB (1.26%)
64  Matrix4: 2.5MB (1.2%)
65  Object { context, memoizedValue, next }: 2.5MB (1.19%)
66  Euler: 1.9MB (0.93%)
67  Quaternion: 1.8MB (0.89%)
68  Object { handlers, memoizedProps, previousAttach, root, type, ... }: 1.7MB (0.82%)
69  LineCurve: 1.7MB (0.81%)
70  HyperInstancingPlaceholder: 1.6MB (0.8%)
```

```typescript
): {type: string; size: number}[] {
    if (!node) return [];

    const memoizedStateNode = node.references.find((ref) => ref.name_or_index === "memoizedState")?.toNode;
    const nextNode = node.references.find((ref) => ref.name_or_index === "next")?.toNode;
    if (memoizedStateNode) {
        return [
            {type: types[i] ?? "unknown", size: memoizedStateNode.retainedSize},
            ...walkHookChain(nextNode, types, i + 1),
        ];
    }

    return [];
}

snapshot.nodes.forEach((node) => {
    if (node.name === "FiberNode") {
        let componentName = node.references.find((ref) => ref.name_or_index === "type")?.toNode.name;
        if (!componentName) return;
        if (componentName === "Object") {
            const typeofNodeId = node.references
                .find((ref) => ref.name_or_index === "type")
                ?.toNode.references.find((ref) => ref.name_or_index === "$$typeof")?.toNode.id;
            componentName = `@${typeofNodeId}`;
        }

        const record = componentMemMap.get(componentName) ?? {
            instances: 0,
            total: 0,
            shallow: 0,
            memoizedState: 0,
            perHook: [],
            memoizedProps: 0,
            children: 0,
            sibling: 0,
        };

        record.instances += 1;
        record.total += node.retainedSize;
        record.shallow += node.self_size;

        const _debugHookTypesNode = node.references.find(
            (ref) => ref.name_or_index === "_debugHookTypes",
        )?.toNode;
        const types: string[] = [];
        if (_debugHookTypesNode) {
            for (let index = 0; index < 1000; index++) {
                const element = _debugHookTypesNode.references.find(
                    (ref) => Number(ref.name_or_index) === index,
                )?.toNode;
                if (!element) break;
                types.push(element.name);
            }
        }

        const memoizedStateNode = node.references.find((ref) => ref.name_or_index === "memoizedState")?.toNode;
        if (memoizedStateNode) {
            record.memoizedState += memoizedStateNode.retainedSize;
        }
        const memoizedPropsNode = node.references.find((ref) => ref.name_or_index === "memoizedProps")?.toNode;
        if (memoizedPropsNode) {
            record.memoizedProps += memoizedPropsNode.retainedSize;
        }
        const childrenNode = node.references.find((ref) => ref.name_or_index === "children")?.toNode;
        if (childrenNode) {
            record.children += childrenNode.retainedSize;
        }
        const siblingNode = node.references.find((ref) => ref.name_or_index === "sibling")?.toNode;
        if (siblingNode) {
            record.sibling += siblingNode.retainedSize;
        }
```

1. Find all the FiberNode data structures in memory
2. Determine which React component they belongs to
3. Compute statistics about that FiberNode
4. Accumulate all the computed statistics, grouping them by React component type

```
SuperDuperPathInstance:
  instances: 671
  total: 1.1MB
  shallow: 88.5KB
  total children: 0 byte
  total sibling: 18.7KB
  total memoizedProps: 190.4KB
  total memoizedState: 864.5KB
  per hook:
   [0] 40.2KB (useMemo)
   [1] 40.2KB (useMemo)
   [2] 53.6KB (useMemo)
   [3] 53.6KB (useMemo)
   [4] 48.3KB (useMemo)
   [5] 69.7KB (useMemo)
   [6] 10.7KB (useRef)
   [7] 10.7KB (useRef)
   [8] 42.9KB (useMemo)
   [9] 67.1KB (useMemo)
   [10] 75.1KB (useMemo)
   [11] 40.2KB (useMemo)
   [12] 96.6KB (useLayoutEffect)
```

```
Portal:
  instances: 696
  total: 4.1MB
  shallow: 91.8KB
  total children: 0 byte
  total sibling: 19.4KB
  total memoizedProps: 183.8KB
  total memoizedState: 353.5KB
  per hook:
   [0] 261.6KB (useContext)
   [1] 30.6KB (useState)
   [2] 61.2KB (useState)
   [3] 879.6KB (useCallback)
   [4] 62.6KB (useState)
   [5] 61.2KB (useEffect)
```

```
AbstractRouteSegment:
  instances: 1340
  total: 277.6KB
  shallow: 176.8KB
  total children: 0 byte
  total sibling: 37.5KB
  total memoizedProps: 112.5KB
  total memoizedState: 4.5MB
  per hook:
   [0] 85.7KB (useMemo)
   [1] 85.7KB (useContext)
@715773:
  instances: 19828
  total: 27.1MB
  shallow: 2.6MB
  total children: 0 byte
  total sibling: 999.5KB
  total memoizedProps: 5.5MB
  total memoizedState: 19.3MB
  per hook:
   [0] 160 bytes (useContext)
   [1] 160 bytes (useRef)
   [2] 704 bytes (useRef)
   [3] 768 bytes (useImperativeHandle)
```

How many **strings** are UUIDs?

```
let totalSize = 0;
let uuidSize = 0;
snapshot.nodes.forEach((node) => {
    if (node.type === "string") {
        const matchesUuid =
            /^[0-9a-fA-F]{8}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{4}\b-[0-9a-fA-F]{12}.*$/g.exec(
                node.name,
            );
        if (matchesUuid) {
            // console.log(node.name);
            uuidSize += node.self_size;
        }
        totalSize += node.self_size;
    }
});

info.topLevel(`Total size of strings: ${utils.getReadableBytes(totalSize)}`);
info.topLevel(`Total size of UUIDs: ${utils.getReadableBytes(uuidSize)}`);
```

Memory Analysis is **Difficult**

But, for some apps, it makes the difference

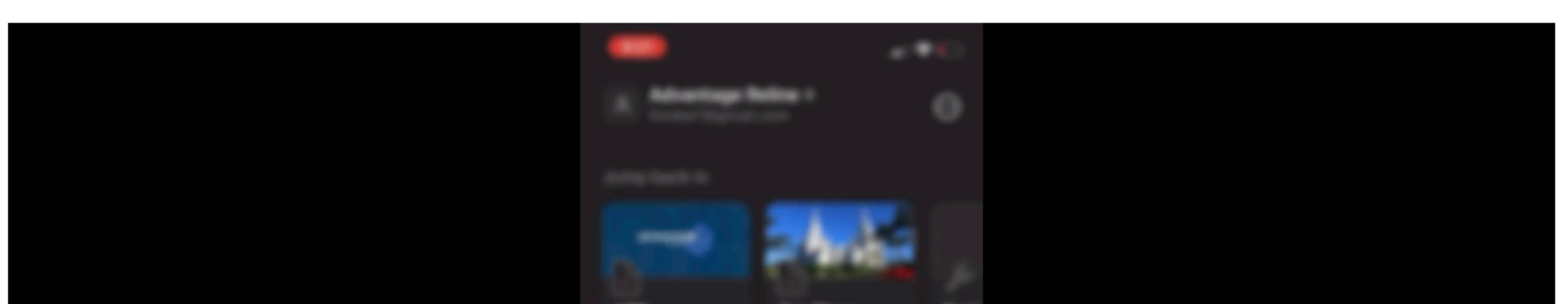
Postato da u/fordee7 4 mesi fa

6

Notion mobile app and browser crash reboot loop - unusable

Request/Bug

Notion crashes (goes into a crash/reboot loop) on mobile app and browser with a database of more than about 50-75 rows/pages. We've tested this on multiple phones. All do it - although some work ok when others are crashing. Then tomorrow the users that worked yesterday, crash today. Works good on a computer. This is unacceptable. I've submitted support tickets and so far all i get is a response that they had issues yesterday or the other day but resolved them. I have a team of 6 people having this issue. So bad it is unusable on mobile.



The Chrome Profiler is cool…
…but sometimes is **not enough**

Constructor	Distance	Shallow Size		Retained Size ▼	
▼ Object ×2254628	2	62 534 152	17 %	199 263 952	56 %
▶ Object @6415517	9	12	0 %	16 939 416	5 %
▶ Object @5447681	4	12	0 %	10 868 616	3 %
▶ Object @3086099	6	28	0 %	5 190 988	1 %
▶ Object @3083509	6	12	0 %	3 277 032	1 %
▶ Object @2908085	6	12	0 %	2 174 664	1 %
▶ Object @4066881	9	164	0 %	1 282 884	0 %
▶ Object @4066883	9	12	0 %	1 280 500	0 %
▶ Object @4842229	8	36	0 %	1 225 608	0 %
▶ Object @12782069	9	28	0 %	1 224 440	0 %
▶ Object @4067157	7	196	0 %	1 221 428	0 %
▶ Object @5338541	7	16	0 %	1 190 544	0 %
▶ Object @5345999	10	12	0 %	1 181 120	0 %
▶ Object @1179777	19	28	0 %	1 095 972	0 %
▶ Object @10423241	10	12	0 %	1 088 576	0 %
▶ Object @3150795	6	12	0 %	935 548	0 %
▶ Object @5342799	7	40	0 %	888 980	0 %
▶ Object @5340719	10	12	0 %	882 620	0 %
▶ Object @7330603	5	24	0 %	866 176	0 %

Thank you!