Debugging Memory Issues in JavaScript

Use Chrome DevTools to fix performance issues and memory leaks in your code

How memory issues can impact performance and end-user experience

When your code has memory leaks or you try to use more memory than is available for your program, it can cause websites to slow down and crash. Accidentally introducing memory leaks into your application can be as easy as either of these scenarios:

- Adding an event listener but never removing it
- Adding variables to the global object

When you set objects as data on a DOM element, it can lead you to use more memory than is available, or can even lead to a memory leak if you never remove the DOM object. (For example, you might render a modal box off-screen and keep it there.) When that happens in production, your end users experience the impact.

To find the source of memory issues, we have to consider questions like:

- From where was memory allocated?
- Why wasn't some piece of memory garbage collected?
- How is memory usage growing over time?

Browser tools provide options for answering these questions.

Developer Tools that can help you debug memory issues

Most browsers, including Google Chrome, Firefox, and Safari, have built-in tools you can use to debug and test your code. While those tools all have feature parity, here we'll look specifically at the Memory Panel in Chrome's Developer Tools, "DevTools", which provides information on how memory is being used.

There are a few different tools available in the Memory Inspector:

- **Heap Snapshot**: This tool shows you how memory is distributed among a page's JavaScript objects and related DOM nodes.
- **Allocation instrumentation on timeline**: This tool shows JavaScript memory allocations over time, and can be used to isolate memory leaks.
- **Allocation sampling**: We can use this to record memory allocations using a sampling method. This tool is best for long-running operations.

Heap Snapshot

Heap snapshots are great to use when debugging memory issues, because you can see what is taking up memory at a given time and compare between times. This view shows:

- What objects are in memory
- Details about the objects that are in memory, like the constructor function that was used to make the object
- How objects reference each other
- The size of memory objects are using
- What's been added or failed to be garbage collected between snapshots

The Heap Snapshot tab has 4 different views:

- **Summary view**: This view groups objects by constructor and can be used to track down DOM leaks.
- **Comparison view**: This displays the difference between Heap snapshots. So if you use more memory or have memory garbage collected between snapshots, you can identify if you have a memory leak based on reference counts and change in freed memory. Note that this view will only be available if you have 2 or more heap snapshots saved in the DevTools Memory Panel.
- **Containment view**: This view is helpful for analyzing objects referenced in the global namespace (Window) to track down why they are not being garbage collected.
- **Statistics view**: This shows a breakdown of how memory is being used based on various categories, like Code, Strings, and JS arrays.

Allocation Instrumentation on Timeline

You can use the allocation timeline to identify objects that aren't being garbage collected when they should be and therefore remain in memory. It is a combination of details you'd find in the heap snapshot, but with the addition of timeline tracking that takes updated snapshots as frequently as every 50 ms. You can use the allocation timeline by making a recording and performing page actions while the recording is underway.

This view shows:

- Where objects were allocated during code execution
- How often garbage is being collected

Allocation Sampling

Allocation sampling is similar to the allocation timeline, but has lower performance overhead. Snapshots are taken as a sample of the stack trace rather than with a regular cadence as they are when using the allocation timeline. You can use allocation sampling when you need to record snapshots for long-running operations and use the stack trace to identify where allocations are originating.

This view shows:

- What lines of code are creating garbage for the garbage collector
- What lines of code are allocating new heap memory

Chrome Dev Tools Lab: How to use the Memory Inspector

Popular browsers have developer tools built-in. In this section, we'll walk through how to use Chrome's memory-related developer tools to inspect memory and performance. To follow along, make sure you've got the Chrome browser installed, and open a browser window. You may also choose to follow along using the browser of your choice, if you prefer: Firefox, Apple Safari, Opera, or Microsoft Edge.

Part 1: Open DevTools to allocate and inspect memory

Step 1

Open **lab1-1.html** in your browser. You should find a button with the text "Allocate Memory". The code adds an "Allocate Memory" HTML button to the DOM and uses an event listener to add a string of an arbitrary length to the `strings` array each time the button is selected. We can use the memory inspector to see how memory allocation varies depending on the size of the string.

There are other buttons — "Allocate A Large Amount of Memory," which calls `stringBuilder()` 1,000 times, and "Allocate Too Much Memory," which calls `stringBuilder()` 10,000 times.

Note: The final button will likely crash your browser, so don't select that button until the later part of the lab that instructs you to select it! You can always reopen the file in the browser if you need to return to the lab.

Step 2

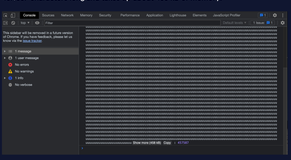
To open Chrome's DevTools, select View > Developer > Developer Tools. You can also use keyboard shortcuts:

- On Mac: Command + Option + I
- On Linux/Windows: F12 or Control + Shift + I

Step 3

Select the "Allocate Memory" button at least one time to trigger the `stringBuilder()` function from the `lab1-1.html` file.

In the Console tab, a string generated by the `stringBuilder()` function is logged, followed by the number of characters in the string. This view shows the approximate size of the string in parentheses. In the example below, the string is 457,587 characters long and takes up about 458 kB of memory:



Note: You can click on images throughout this article to view them at a larger size.

Part 2: Use the Memory tab to see what JavaScript objects are taking up memory by reading a heap memory snapshot

You can use the Heap snapshot tool to profile the heap and find memory leaks.

Step 1

Open the Memory tab in the Inspector and select the Profiles panel on the left.

Step 2

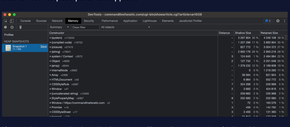
Select the Heap snapshot radio button.

Step 3

Select the round button near the top left of the Profiles panel to create a snapshot. (An alternative option is to scroll to the bottom of the Profiles "Select profiling type" page and to select the blue "Take snapshot" button.)

Step 4

Select the Snapshot in the list of Heap Snapshots from the side bar. A summary view of the Heap Snapshot will appear:



If you take a closer look at the image, you'll see that many JavaScript objects were instantiated to make the page load, including some JavaScript engine internals, like `(compiled code)` and `(system)`. Generally, you can ignore anything in parentheses and assume that leaks will be coming from the code you are running and not the browser source code.

In the summary view, objects are grouped based on the constructor function that was used to instantiate an object. In our case, we used the `stringBuilder` constructor function to instantiate `stringBuilder` objects.

Step 5

Scroll down the list until you see `stringBuilder`. Take note of some of the other objects and see what properties are available on the objects to get more familiarity with the summary view.

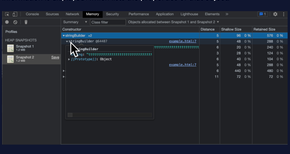
You can filter objects using the search bar at the top of the Memory Inspector.

Step 6

Locate the search bar and filter for `stringBuilder` objects.

Each of the `stringBuilder` objects is listed with a unique number. Those numbers are only there for debugging purposes to be able to identify unique instances of objects — the numbers do not correspond to memory addresses. You can view it as `stringBuilder @64487` in the image:

If you hover over a `stringBuilder` instance, a snapshot of debugging information is displayed, which lists the properties of that object:



Step 7

Expand an instance of `stringBuilder` under the `stringBuilder` class list.

Step 8

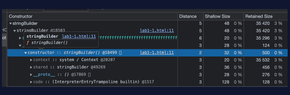
Take note of what you see: This object has a `string` object and a built-in `__proto__` object. On the right-hand side, there are columns for Distance, Shallow Size, and Retained Size.

Step 9

Expand the `__proto__` property, then expand the `constructor` property. You can see the constructor function `stringBuilder()` was used to instantiate the object. When debugging, this can be helpful to identify how an object was instantiated if you are trying to track down objects that are being maintained in memory.

Step 10

Hover over the `constructor` property. You'll see a hyperlink to `lab1-1.html:11` and can hover over the `stringBuilder()` function to see the function definition. The 11 at the end of `lab1-1.html` indicates that the function definition begins on line 11 — that's helpful to know in case you are working with a large code file and need to quickly find the right line of code to review.



Step 11

Select the `lab1-1.html` link. The file will open in the Sources tab. Find the `stringBuilder()` function definition on line 11.

Step 12

Select the Memory tab to return to the previous tab. Next, we'll dig into what each column in the Memory panel is indicating.

- **Distance**: Using the shortest path of nodes from the Window/root, this is the count of nodes on that path.
- **Shallow Size**: Shallow size is the size in bytes of the object's internal memory. Often, strings and external arrays are in memory in *renderer memory*. Renderer memory is a combination of the native memory, JavaScript heap memory of the page, and JavaScript heap memory of all of the workers that were started by the page.
- **Retained Size**: Retained size is the maximum retained size in bytes of a single object from objects of the same class. Once an object's memory can be freed, this is the memory that is freed.

Step 13

How much shallow memory is being used by all instances of the `stringBuilder()` function? Make a note of this value. When debugging, you can check if memory of an object is increasing over time by checking the size of the object at various heap snapshots over time.

Step 14

Identify how much shallow memory is being used by a single instance of a `stringBuilder` object.

Let's take a look at how the sum of the shallow memory can increase. When the amount of memory is increasing, we know that it's from a combination of our existing objects growing, new objects being created, and existing objects potentially not being garbage collected. If memory increases when we're not expecting it, it can indicate a memory leak.

Step 15

Select the Allocate Memory button 5 times.

Step 16

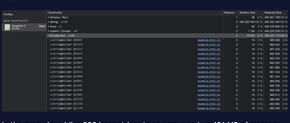
Take a new Heap snapshot.

Step 17

Identify any changes in the Shallow Size or Retained Size columns for the `stringBuilder()` class and `stringBuilder` instances. Would it be a problem if we have thousands of `stringBuilder` instances in memory?

Step 18

Select the "Allocate A Large Amount of Memory" button. Take a snapshot and review how much memory was used. The snapshot will list a MB total in the Heap Snapshots summary list.

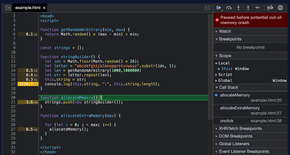


In the example, adding 500 large strings to an array used up 496 MB of memory. In general, your goal should be to minimize memory usage, and 496 MB is pretty high for an app that builds strings.

While there's no magic memory number, for a frame of reference — if you're building a Todo List application and it takes up 1 GB of memory, that is extremely high and there's likely room for optimizations, whereas if you're building a video game and it takes up 1 GB of memory, that would be really great and likely wouldn't need optimizations.

Step 19

Select the "Allocate Too Much Memory" button. If you switch to the console tab, you can see approximately how many times the function is able to execute before the browser tab crashes. What error message do you expect to see?



```
<body>
<button onclick="allocateMemory()">Allocate Memory</button>
<button onclick="allocateMemoryAndAddToGlobal()">Allocate A Large Amount of Me
<button onclick="allocateLeftMemory()">Allocate Too Much Memory</b
</body>
</html>
```

Free response

Open the **debugging-memory-01.html** file in a new tab.

The code here adds 1,001 elements to the page: 1 'parent' element and 1,000 additional elements.

Initially, all of the child elements have `position:relative` positioning.

When you click the "Move elements" button, half of the elements are converted to have `position: absolute`. The other half are moved with a computed value for the `left` property.

Back Next

button at least 10 times to capture the jar numbers mentioned.

Based on what you've learned and information found using your browser's Developer Tools, answer the following questions:

- What warnings are you likely to encounter when running this code?
- What type of design decisions can you make to avoid this type of result and improve performance?

Submit Response

Stuck? Get a Hint

Part 3: Guided Debugging of a Memory Issue

Step 1

Open **lab1-2.html** in your browser. The page has two buttons.

```
<html>
<head>
</head>
<body>
    <button id="detachDomElements">Detach</button>
    <button id="deleteDomElements">Delete</button>
</body>
</html>
```

Step 2

Open the inspector to the Console tab. Paste in the following code.

```
let detachedElements;

function create() {
    let ul = document.createElement('ul');
    for (let i = 0; i < 100; i++) {
        var li = document.createElement('li');
        ul.appendChild(li);
    }
    detachedElements = ul;
}

function removeEl() {
    detachedElements = null;
}

document.getElementById('detachDomElements').addEventListener('click',
create);
document.getElementById('deleteDomElements').addEventListener('click',
removeEl);
```

You'll see functions for creating detached DOM elements and for removing the elements that are used as callbacks for `click` event listeners on the "Detach" and "Delete" buttons. We refer to them as detached DOM elements because they've been created using `document.createElement()`, but are not appended to the document yet.

In the case of this code snippet, the memory is retained because a reference to the elements is made with the `detachedElements` variable, even though they're not actually appended to the document at any point. We resolve the issue by setting the `detachedElements` variable to null using the `removeEl()` function.

Walk through the steps to identify what a detached node looks like in a heap snapshot when a detached node exists.

Step 3

Select the Detach button.

Step 4

Open the Memory tab and create a Heap snapshot.

Step 5

Type "Detached" into the filter bar to show Detached DOM elements. Review the results. You should see 1 detached `<ul>` element and 100 detached `<li>` elements.

Let's remove the detached elements to fix our memory issue.

Step 6

Select the Delete button.

Step 7

Create a new Heap snapshot.

Step 8

Type "Detached" into the filter bar. No results should appear because we set the `detachedElements` variable to `null` using the `removeEl()` function.

Part 4: Use the Performance Tab to measure memory usage over time in an application

Next, we'll dive into how you can analyze the runtime performance of your applications and locate a potential performance bottleneck. Even if your site loads quickly, it can have laggy performance issues while it is being used, so it's important to test a site after it's fully loaded with simulated activity.

Note: To ensure a clean environment during the lab, use your browser's private browsing mode at all times or clear your browser cookies and remove all extensions. In production code, you'd want to test with a normal environment as some performance regressions can be caused by interactions with browser extensions. For example, if you have a specific extension installed and it causes unexpected behavior by your application, it can be useful to log information to reproduce the issue.

This part of the lab uses the **lab2.html** file. This file renders buttons that can be selected to initialize the code we use and then to add and clear space ships.

Here's what's going on in the code:

- The code itself includes a `createApp()` function that executes when you select the "Start" button.
- The `createApp()` function initializes a space ship count and grabs the size properties of the `document.body` using the built-in `getBoundingClientRect()` `Element` method.
- `createApp` objects have a method `makeShips()`, which uses the `height` from the HTML `body` element to set various start values for the `top` property to place space ships on the visible page.
- The method then creates a custom `@keyframes` definition for each spaceship to make them spin at random rotation speeds by setting a randomized animation speed of up to 20 seconds.

So each time a new spaceship is created, the following actions occur:

- A new `@keyframe` rule is added to the stylesheet.
- A new spaceship is appended to the `body`.

Two more buttons appear, an "Add" button and a "Clear" button. The "Add" button adds another 10 spaceships to the page. You can use the "Clear" button to revert back to 10 spaceships if you slow down the performance too much or want to reset your analysis during the lab.

Step 1

Open the browser and drag the **lab2.html** file into the window.

Step 2

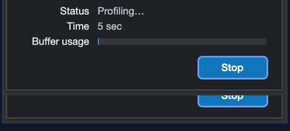
Select the "Start" button. Ten space ships will be added to the DOM.

Step 3

Open DevTools and navigate to the *Performance* tab. The Performance tab can be used for auditing performance of a page. It provides a detailed overview of details such as frames per second and CPU performance with snapshots of the page at each point in time, and has built-in features to simulate throttling the network and CPU caps to match slower environments.

Step 4

Select the circle button at the top left of the tab to start a Performance profile. A popup will appear to indicate that the Performance profile is recording web activity, and has a button to end the recording.



Step 5

Select the "Add ships" button on the page.

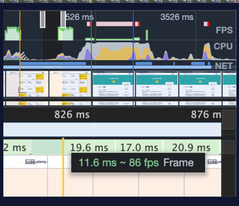
Step 6

Run the profiler for approximately 1 minute and continue to select the "Add Ships" button every few seconds until you notice any latency or performance impact on your machine. When that starts to happen, select it a couple more times, then select "Stop" to end the profile. The goal is to cause poor performance to see what it looks like in the Performance tab.

Step 7

Select the "Clear Ships" button. In the profile summary view, begin by looking at the *frames per second (fps)*, which is the frequency that new frames are displayed. An ideal level is 60 fps — if the fps is below 60, it means that frame rendering is too slow and the screen can appear janky. You can locate fps at the top of the chart. The levels are indicated with a green space chart and an overlapping red bar chart.

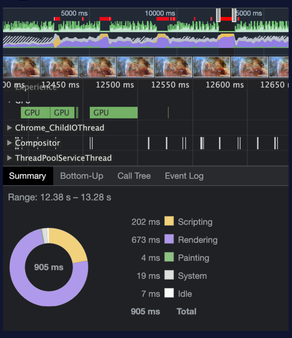
In the following two images, you can see the difference on the timeline between high and low frames per second. The first chart has low fps, and the fps timeline is red. Hovering over the timeline reveals that not only are the fps around 2 fps, but the frame was dropped. Dropped frames results in a poor user experience. In the second image, the fps timeline is green and hovering over the timeline reveals that it's around 86 fps, which is a good fps rate.



This timeline helps you identify where a performance bottleneck occurred during code execution. If you see red, it's likely there is an issue to investigate.

portion of the chart that you are analyzing to get a closer look at specific parts of the timeline (or to get a wider view). There are also keyboard shortcuts available:

- `w` - Zoom in
- `a` - Scroll back
- `s` - Scroll forward
- `d` - Zoom out



In the image, the timeline is zoomed in on a portion of the timeline that had low fps performance and it indicates that a majority of the CPU was used for scripting and rendering. In general, your goal should be to reduce the amount of rendering/work done per frame to prevent jank.

Step 9

Under the charts, open the "Main" dropdown to view the main thread activity flame chart. In this chart, the y-axis is the call stack, and the x-axis is the recording over time. The stacked events indicate that an upper event triggered the events below it, and the length of the bar indicates that the event took longer to run. You can zoom in on the timeline so the event names are visible.

If you hover over the main thread events, you'll see how long an activity took to run, and might see some warning messages indicated in red. If an event has a red arrow in the upper right corner, it means that there could be an issue related to the event.

Here are a few warning message you can find from this lab:

- `Warning Forced reflow is a likely performance bottleneck`
- `Warning Long task took 941.27 ms`
- `Warning Handler took 829.27 ms`

In this example, the purple "Recalculated Style" activities have a `Forced reflow` warning. A forced reflow event occurs when the browser has to recalculate the size and location of elements on your page. (This process is called *Layout* in Chrome, Opera, Safari, and Internet Explorer, and *Reflow* in Firefox. Layout can happen due to code that triggers style updates. This can cause a performance bottleneck because the layout update is synchronous. In some cases, triggering Layout can have minimal impact, so the warning doesn't always mean there is an issue in your code.)

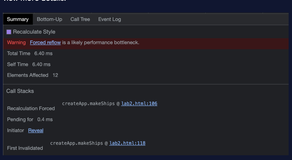
When you see this warning, consider:

- How many elements require layout?
- How complex are the layout changes?
- How long does Layout take to run? (by checking with Chrome DevTools)
- Are there any noticeable performance issues, such as lag?
- Are there performance issues when testing with a throttled environment?

The solution for this type of issue is to first read styles, and separately set new styles to avoid as much lag, or to use modern CSS layout models like flexbox.

Step 10

Select one of the "Recalculated Style" blocks to open the event summary and view more details.



Many details are listed in the Summary tab:

- Total Time
- Elements Affected
- Call Stack details

Step 11

Under the call stack details, select the link for `createApp.makeShips @ lab2.html:106` to open the **lab2.html** file in the "Sources" tab. The file will be annotated with how long it took to execute each line.

```
100
101         createApp.prototype.makeShips = function() {
102 0.6 ms    let save = document.querySelector('.ship').cloneNode();
103 16.2 ms   document.querySelectorAll('.ship').forEach(function(d){
104 8.1 ms    x.remove()});
105 8.2 ms    for (let i = 0; i < this.count; i++) {
106 147.5 ms    this.bodySize = document.body.getBoundingClientRect()
107 13.9 ms       let ship = save.cloneNode();
108 8.2 ms        ship.style.left = (i / (this.count / 100)) + 'vw';
109 7.3 ms        ship.classList.add('hashline');
110 2.5 ms        ship.style.top = Math.random() * this.bodySize.height;
111 21.7 ms       ship.style.top = top + 'px';
112 1.3 ms        let name = 'customRule1top';
113                 let time = Math.floor(Math.random() * 20);
114 2.5 ms        let rule = `@keyframes ${name} {
115                 from { transform: rotateZ(0deg); }
116                 to   { transform: rotateZ(360deg); }
117                 }`;
118 93.4 ms       document.styleSheets[0].insertRule(rule);
119 3.3 ms        let str = `${name} ${time}s linear infinite`;
120 5.6 ms        ship.style.animation = str;
121
122 3.9 ms        document.body.appendChild(ship);
123               }
124             }
```

In the example, line 106 took 147.5 ms to run and line 118 ran for 93.4 ms. In general, 0 to 100ms is ideal for performance as an end user for the result to feel immediate, but perception of performance delays can really vary depending on the network conditions or devices being used. Generally, mobile devices have less CPU than laptops or desktop computers.

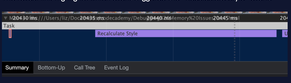
Step 12

Return to the Performance tab.

Step 13

Select the "reveal" link.

The reveal link highlights which event triggered the "Recalculate Style" event.



That wraps up the overview of how to use the Chrome DevTools Performance tab. For further exploration, you can repeat the lab and alter the CPU and Network settings to simulate throttling. Select the gear icon to reveal the settings to change those settings. Try using a 4X CPU slowdown and see how it impacts performance and compare how many ships you can add with and without the 4X slowdown.

Practice working with heap snapshots

Open the **debugging-memory-02.html** file in a new tab. This page contains two buttons.

Triggering the "Task 1" button creates 10,000 customer drink orders in a `for` loop, adds them to an array of objects of orders called `orders1`, and concatenates all of the orders into a string. Notice on `line 34` that `makeDrink()` is called without using the `new` operator, resulting in the `this.order` value being assigned to the `window`.

Triggering the "Task 2" button creates a new `makeDrink2` object which is added to the `orders2` array.

Create three heap snapshots:

- the first, after loading the page
- the second, after selecting the first task button
- the third after selecting the second task button

Answer the following questions concerning **debugging-memory-02.html**:

- How many objects were added after clicking the first button?
- Why might it be a problem for the `order` variable to be added to the `window`?

Submit Response

Back Next

Stuck? Get a Hint

Free response

Answer the following questions concerning **debugging-memory-02.html**:

- Which function call adds `makeDrink` objects to memory?
- Where can you identify their allocation size?
- How many `makeDrink` objects do you expect to find in memory?
- What is the constructor of these objects and on what line of code were they instantiated?

Submit Response

Stuck? Get a Hint

Free response

Answer the following questions concerning **debugging-memory-02.html**:

- Are there any detached elements?
- How can you check?

Submit Response

Stuck? Get a Hint

Summary

Nice work — memory issues can be challenging, especially if you're debugging it while trying to meet a deadline!

In this article, you tried out Chrome's DevTools Memory and Performance tabs and learned how they can be used to identify issues in your code. You also got a chance to hands-on debug some common issues without the pressure you'd experience if you were trying to debug memory for the first time in a professional setting.

In the first lab, we added large strings in memory and explored how that impacts memory allocation using the Heap Snapshot tool. Then we saw how to identify if there are detached elements in memory and how to clear them.

In the second lab, we looked at code that performed poorly using the Performance tab and saw how to identify potential issues in code based on the metrics provided in the performance timeline.

As you develop increasingly complex JavaScript applications, your understanding of how JavaScript memory works and how to debug performance issues will make you a more knowledgeable and valuable JavaScript developer.

Free response

How can you use the Performance and Memory tabs in Chrome DevTools or other browser developer tools to test and optimize your code?

Submit Response

Back Next

Back Next