Web Performance & Debugging

Web performance refers to how quickly a website loads and responds to user interactions. It plays a key role in providing a smooth and enjoyable user experience. A fast website keeps visitors engaged, reduces bounce rates, and boosts search engine rankings.

Debugging is the process of identifying and fixing errors or issues in a website's code. It ensures that the site works correctly across different browsers and devices. Developers use tools to track down bugs, test solutions, and improve overall functionality. Effective debugging helps create a smooth, error-free user experience.

Let's Debug and Make a slow web page Fast

In this module, you'll learn how to debug a backend application and improve its performance.

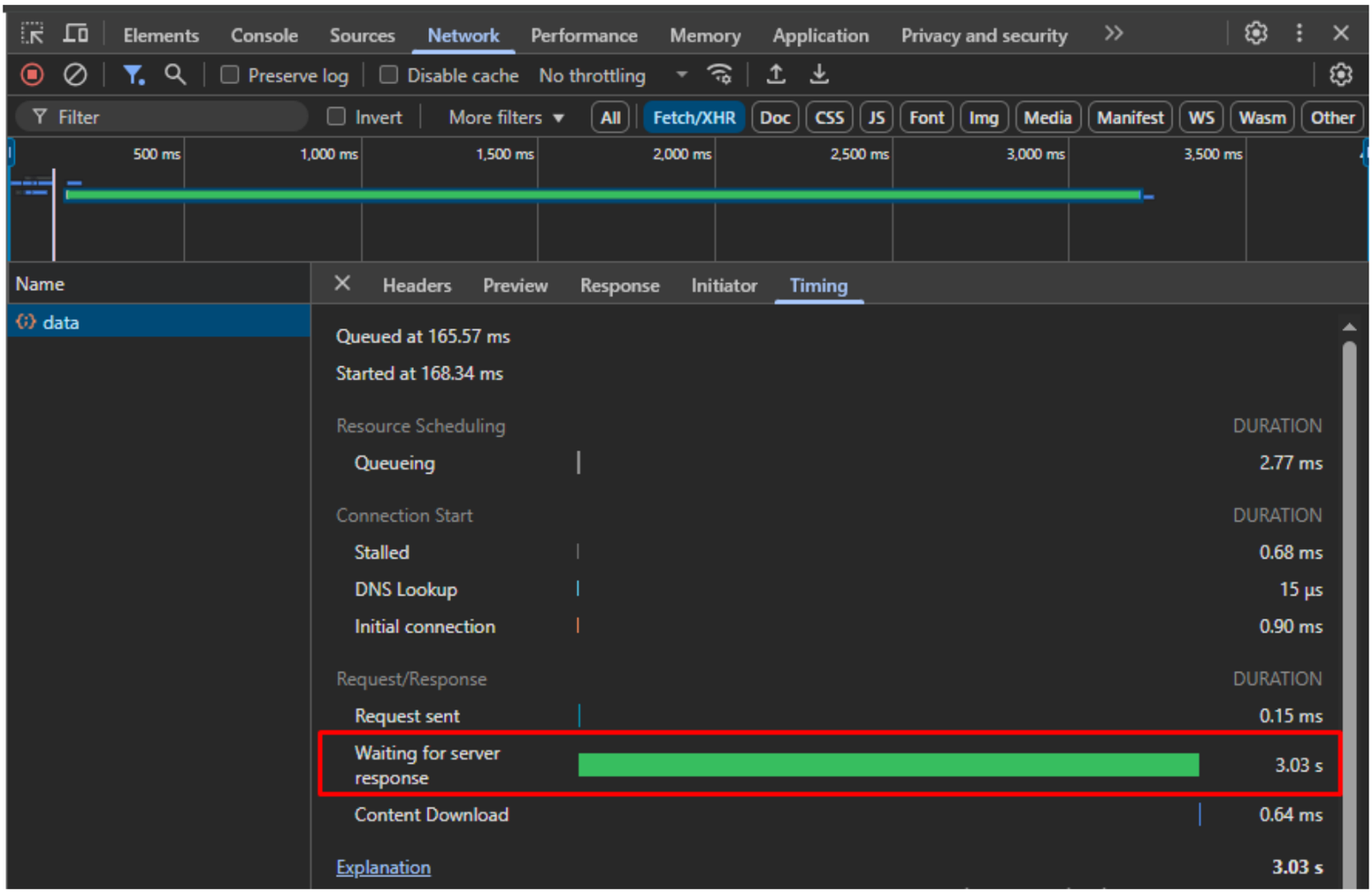
We'll begin with a practical example, which you can explore by downloading the source code from the provided link. The example intentionally includes code that causes performance issues, so you can identify and fix them as part of the learning process.

`click here to for source code`

When you open the project, you'll find both the frontend and backend included. The frontend contains a page that integrates an API from the backend, accessible at the endpoint /api/data.

The performance issue in this project is that the webpage loads slowly, with the server response taking around 3 seconds.
To start debugging, we'll first inspect this from the frontend. Open the browser's Developer Tools, go to the Network tab, and observe the API response from /api/data. You'll see it takes approximately 3 seconds to respond.

While additional debugging can be done on the frontend—such as using the Performance tab to analyze screen paint times—our focus in this module is on the backend, where the actual issue lies.



Now it is clear that page is slow because the response from server is slow. Which confirms that issue is on backend side not on frontend. So we now move to backend and starts our debugging.

Our API Endpoint looks something like this.

```
app.get('/api/data', async (req, res) => {
    console.log('Received request. Querying database...');
    try {
        const data = await orm.getData(db);
        res.json(data);

    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

We'll start with a simple approach by calculating the query time using console.time. This will allow us to measure the time taken for the query to execute and give us a clearer picture of where the delay is occurring. By adding console.time before the query and console.timeEnd after the query, we can track the exact duration it takes to process the request.

```
app.get('/api/data', async (req, res) => {
    console.log('Received request. Querying database...');
    try {
        console.time("cg")
        const data = await orm.getData(db);
        console.timeEnd("cg")
        res.json(data);

    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

```
$ node server.js
Connected to the SQLite database.
Server listening on port 3001
Received request. Querying database...
cg: 3.014s
```

From the console.time output, we can see that the query takes around 3 seconds. This indicates that the issue lies within our query function.

However, this is a naïve approach. As the codebase grows larger, it becomes impractical to manually insert console.time statements throughout the code to trace performance issues. To handle this more efficiently, we use CPU profiling to analyze the performance of the application without modifying the codebase directly.

Methods for CPU Profiling

1. VS Code Debug Terminal
2. Manual Code Insertion (console.profile)
3. Chrome DevTools

1. VS Code Debug Terminal

Taking a Performance Profile in VS Code

1. Open the Debug Terminal:

In VS Code, open the Debug Terminal and start your Node.js application using:

```
node --inspect --server.js
```

2. View the Call Stack:

On the left side of VS Code, you'll see the Call Stack panel, This is where you can track your code's execution flow during debugging.

3. Start the Performance Profiling:

In the Debug Terminal, click on Take Performance Profile.

Once you click on Take Performance Profile, you will be presented with options to choose the type of profiling:
3.1 CPU Profile
3.2 Heap Profile
3.3 Heap Snapshot

4. Select CPU Profiling:
4.1 Manual: Manually start and stop profiling.

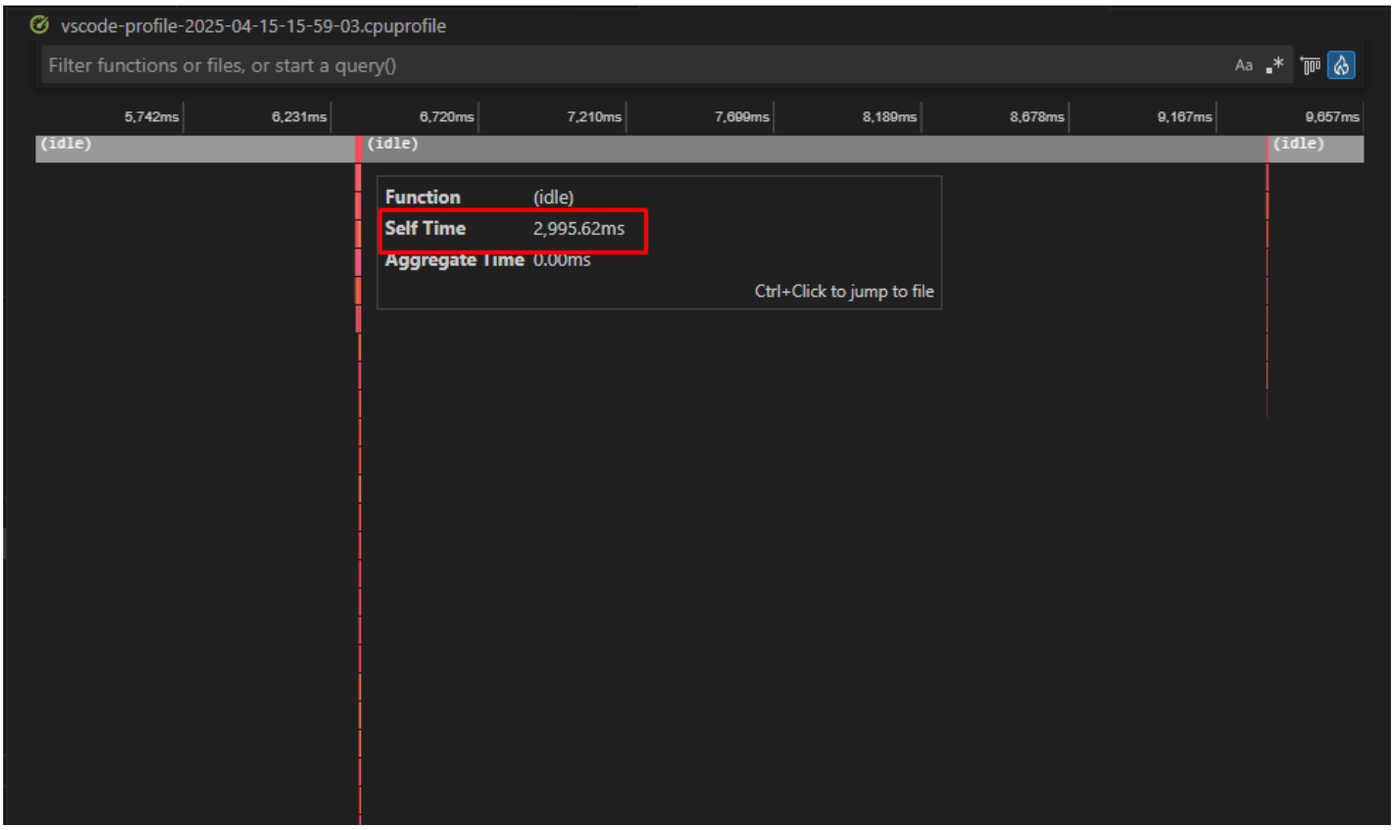
4.2 Duration: Measure the duration of specific operations.

4.3 Breakpoints: Measure time between breakpoints in your code.

For this example, choose "Manual" to start measuring profiling manually.

Once we stop the profiling it will create .cpuprofile file in our folder structure.

Now open this file using flame graph extension and we will see something like this.



As we analyze the performance profile, we can see that the database query, which is an I/O operation, is taking nearly 3 seconds to complete.

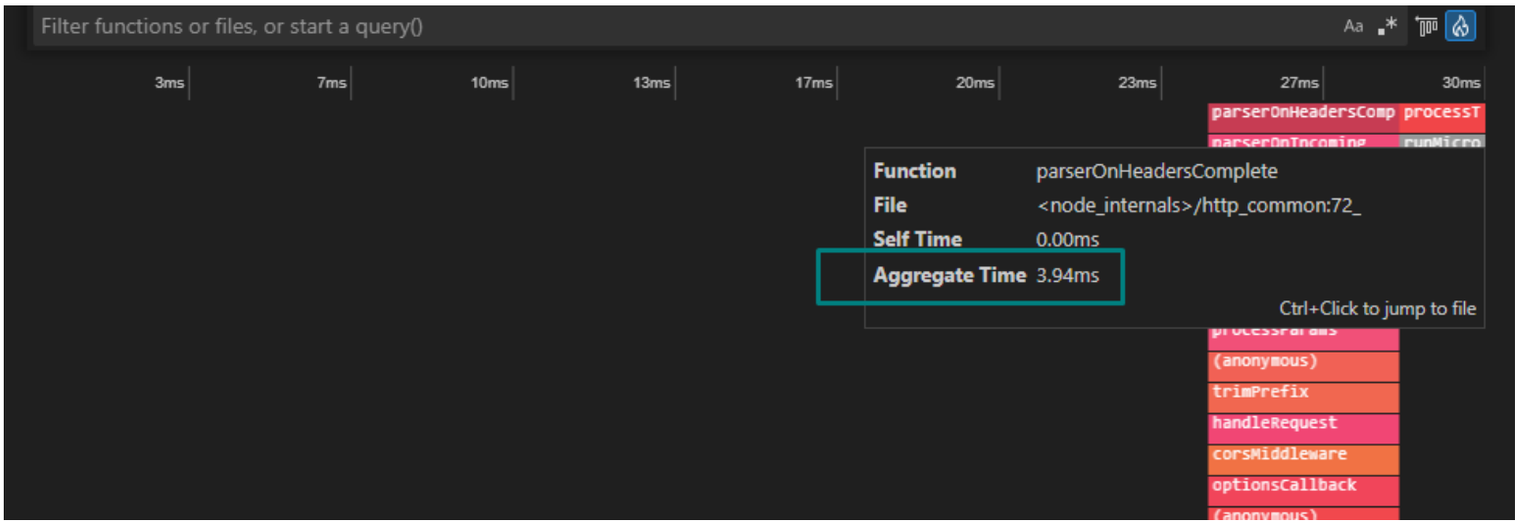
However, during this time, the CPU remains mostly idle, indicating that the CPU is not being heavily engaged while the query is running. This suggests that the bottleneck is not CPU-related but rather due to the slow I/O operation, such as database query latency, which is delaying the overall response time.

Here is the function that do database query and we can see the delay due to setTimeout taking 3 seconds. We will remove this setTimeout and then this issue resolves.

```
const getData = (db) => {

    return new Promise((resolve, reject) => {
        setTimeout(() => {
            try {
                const row = db.prepare('SELECT message FROM messages WHERE id = 1').get();
                resolve({ data: row ? row.message : 'No message found' });
            } catch (err) {
                reject(err);
            }
        }, 3000);
    });
};

module.exports = { getData };
```



The web page will load quickly now

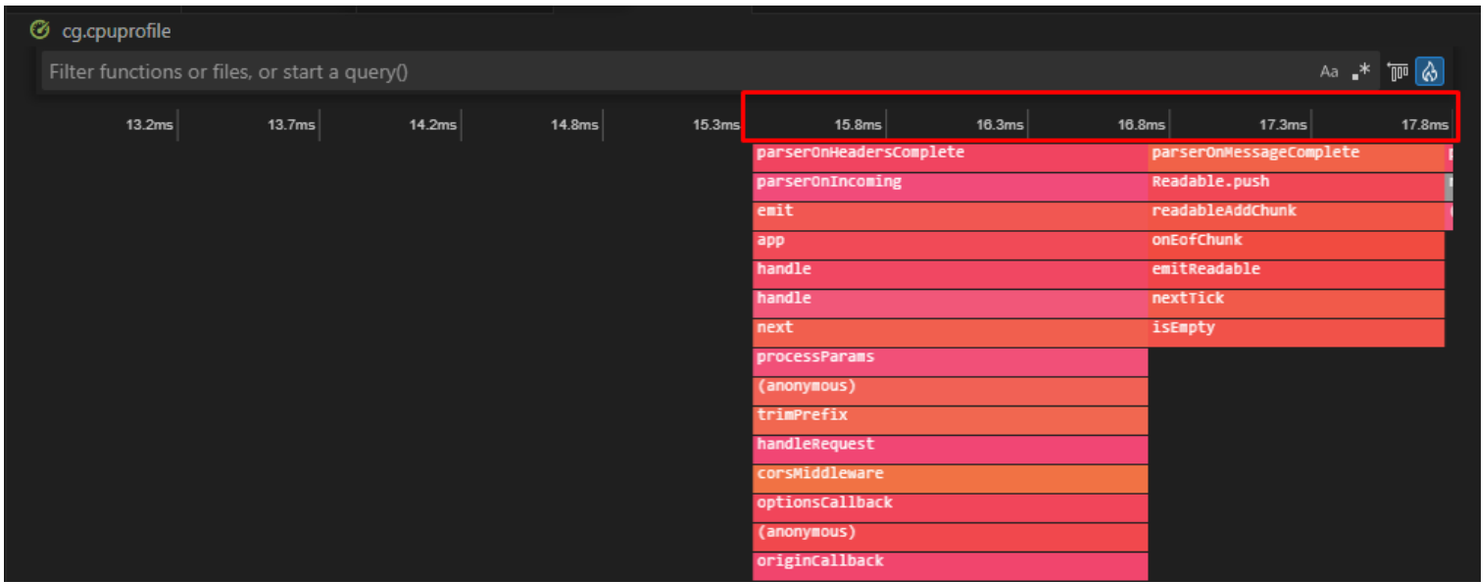
2. Manual Code Insertion (console.profile)

```
app.get('/api/data', async (req, res) => {
console.log('Received request. Querying database...');
 try {
 console.profile("cg")
 const data = await orm.getData(db);
 console.profileEnd("cg")
 res.json(data);
  } catch (err) {
 res.status(500).json({ error: err.message });
   }
    });
```

It will generate cg.cpuprofile file which will have same analysis data

Let's try to block the CPU for 2 seconds and check the profile data

```
function blockForTwoSeconds() {
 const start = Date.now();
 while (Date.now() - start < 2000) {
// Busy-wait loop: do nothing until 2 seconds have passed
  }
 console.log('2 seconds have passed');
    }
app.get('/api/data', async (req, res) => {
console.log('Received request. Querying database...');
 try {
  console.profile("cg")
  blockForTwoSeconds()
  const data = await orm.getData(db);
  console.profileEnd("cg")
  res.json(data);
  } catch (err) {
  res.status(500).json({ error: err.message });
    }
     });
```



In this case, the CPU is blocked for around 2 seconds.
If we have any CPU-intensive tasks that are blocking the main thread, we can offload them to worker threads in Node.js.

This helps keep the main thread responsive and frees up the CPU to handle other tasks, improving overall performance and scalability.

Heap Profiling

Heap Profiling is used to Identifying memory leaks. A memory leak occurs when an application keeps consuming memory without releasing it, even though the data is no longer needed. Over time, this unused memory builds up, causing the app or server to slow down or eventually crash. This is especially problematic in server environments, where memory leaks can continue to grow even if no new users are interacting with the system.

Memory leaks are created usually by global variables. So we need to use them carefully.

To identify the memory leaks we will do a Heap Profiling

After running in VS Code debug terminal we will again run the command

```
node --inspect --server.js
```

Now we click on Take Performance Profile in Call Stack and now we will select the Heap Profile instead of CPU. After selecting Heap profile we will have similar 3 options like Manual, duration or breakpoints. We will choose the manual.

Now to generate memory leak we have declared a global variable and keep adding data in that variable with each request.

```
let id = 0;
const map = new Map()
app.get('/api/data', async (req, res) => {
console.log('Received request. Querying database...');
 try {
      map.set(id++, `value-${id++}`)
      const data = await orm.getData(db);
      res.json(data);
      } catch (err) {
     res.status(500).json({ error: err.message });
               }
                  });
```

To generate memory leaks we will use autocannon, a tool for load testing

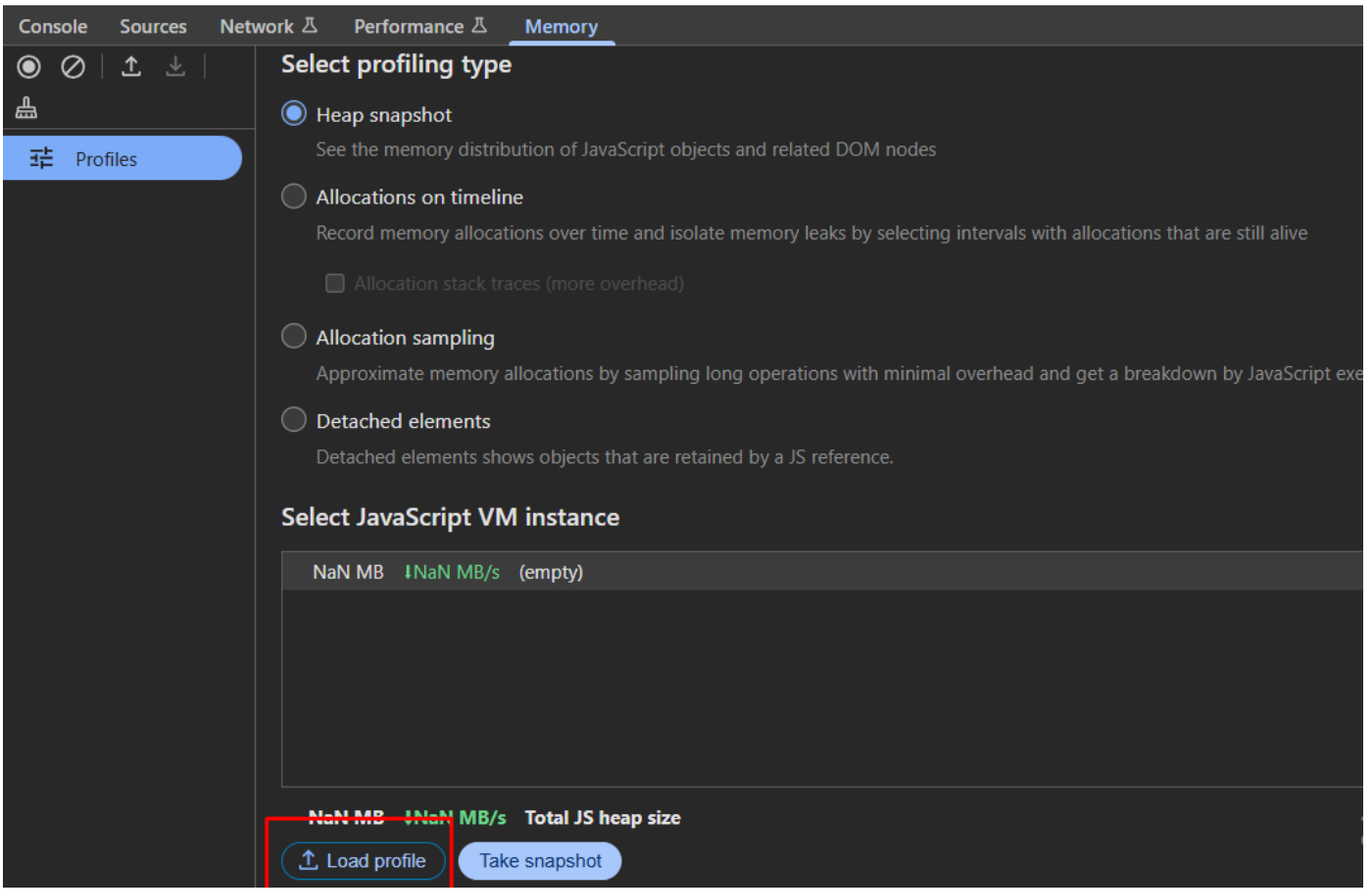
```
autocannon -c 1000 -d 10 -p 10 http://localhost:3001
```

After hitting these concurrent request we will stop the Heap Profiling and now a .heapprofile file will create.

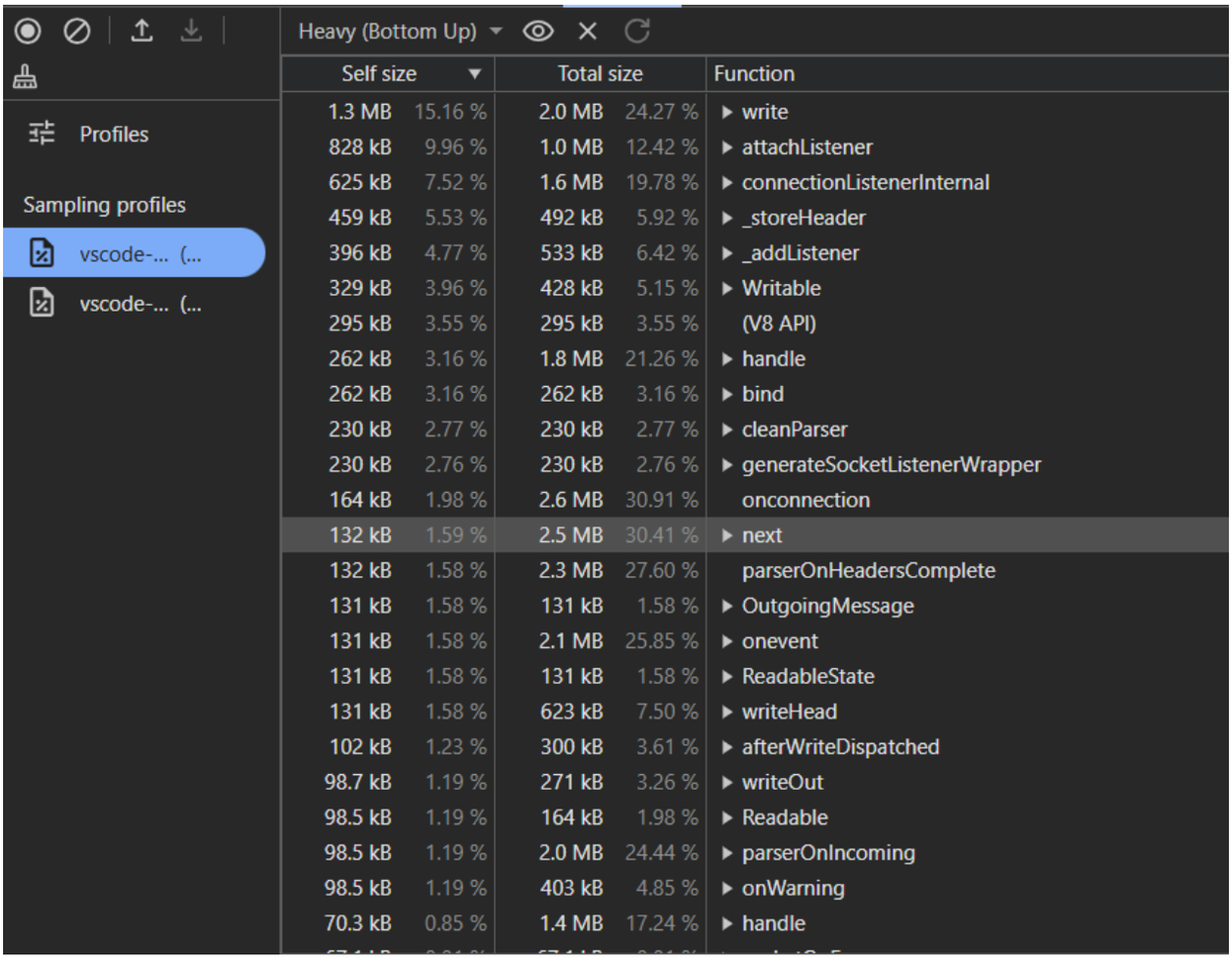
To analyze this file, we will load the .heapprofile in chrome devtools.

For this we will visit to chrome://inspect/#devices

Here on the Memory tab we can load our generated .heapprofile file to analyse.



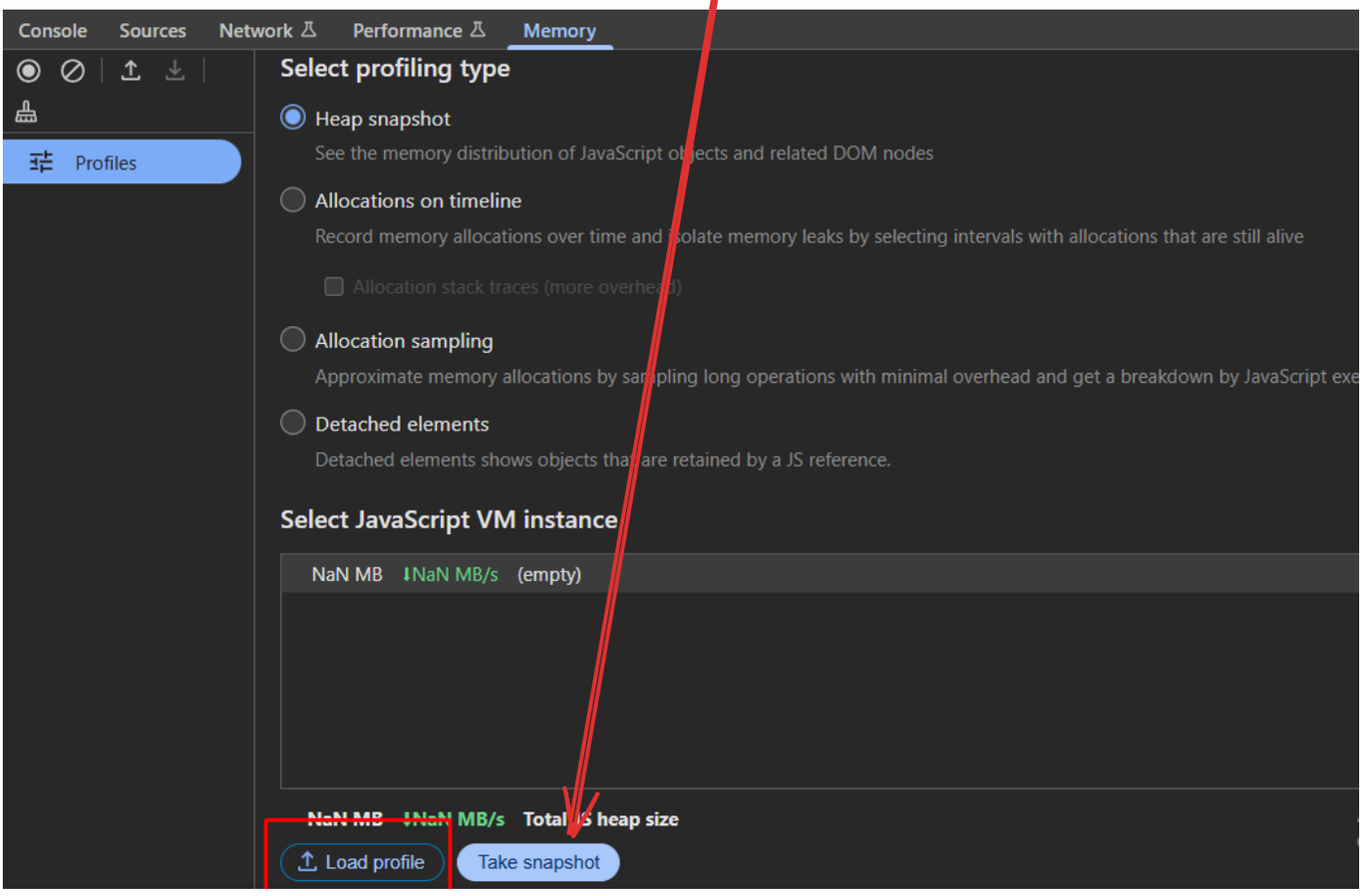
Now we can do the memory analysis which variables and function are consuming more memory



Heap Snapshot

To better understand memory consumption and identify leaks, we can use Heap Snapshots. By comparing two snapshots taken at different points in time, we can track memory usage over time and see where memory is being allocated but not released. This comparison helps pinpoint which objects or areas of the application are consuming excessive memory, making it easier to identify and resolve memory leaks.

We can take heap snapshot directly from the chrome dev tools, since our server is attached there, By clicking Take snapshot button.

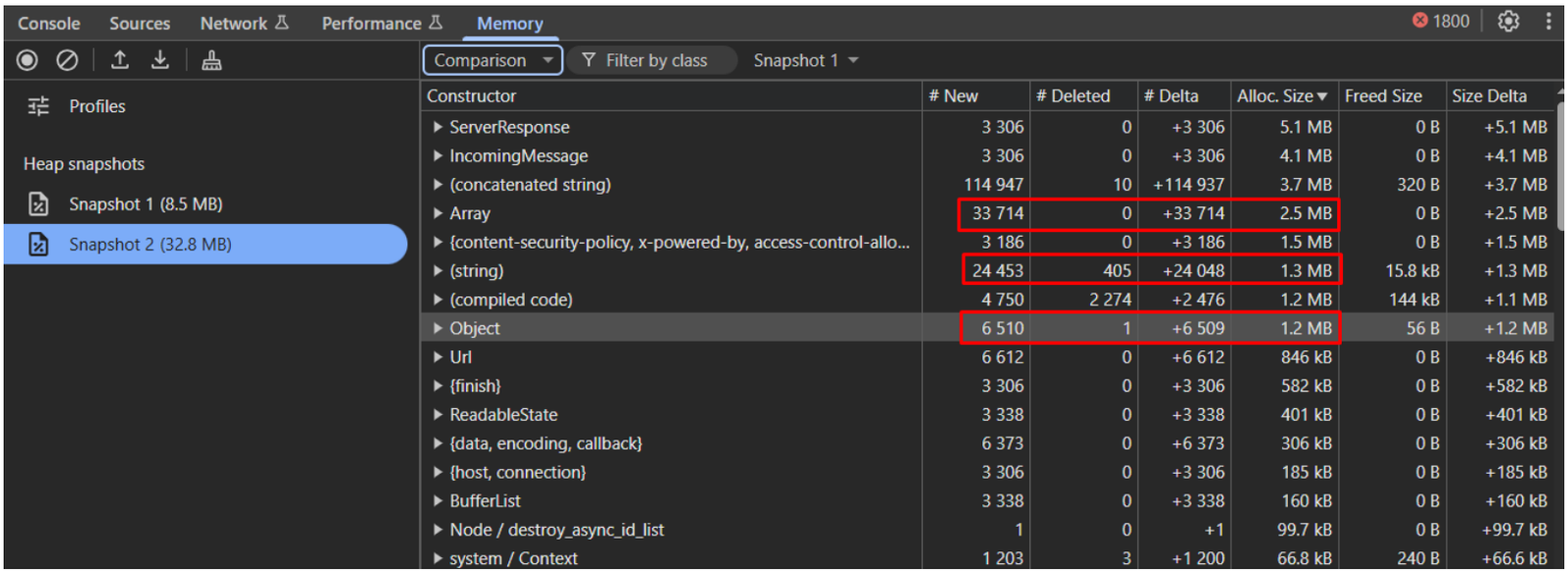


We will take 2 snapshots one before running autocannon and one after running autocannon with command.

autocannon -c 200 -d 100 -p 120 http://localhost:3001

And we will compare both snapshots.

By comparing Heap Snapshot - 1 & Heap Snapshot -2 we can clearly see the difference in memory allocation.



Constructor	# New	# Deleted	# Delta	Alloc. Size	Freed Size	Size Delta
ServerResponse	3 306	0	+3 306	5.1 MB	0 B	+5.1 MB
IncomingMessage	3 306	0	+3 306	4.1 MB	0 B	+4.1 MB
(concatenated string)	114 947	10	+114 937	3.7 MB	320 B	+3.7 MB
Array	33 714	0	+33 714	2.5 MB	0 B	+2.5 MB
{content-security-policy, x-powered-by, access-control-allo...	3 186	0	+3 186	1.5 MB	0 B	+1.5 MB
(string)	24 453	405	+24 048	1.3 MB	15.8 kB	+1.3 MB
(compiled code)	4 750	2 274	+2 476	1.2 MB	144 kB	+1.1 MB
Object	6 510	1	+6 509	1.2 MB	56 B	+1.2 MB
Url	6 612	0	+6 612	846 kB	0 B	+846 kB
{finish}	3 306	0	+3 306	582 kB	0 B	+582 kB
ReadableState	3 338	0	+3 338	401 kB	0 B	+401 kB
{data, encoding, callback}	6 373	0	+6 373	306 kB	0 B	+306 kB
{host, connection}	3 306	0	+3 306	185 kB	0 B	+185 kB
BufferList	3 338	0	+3 338	160 kB	0 B	+160 kB
Node / destroy_async_id_list	1	0	+1	99.7 kB	0 B	+99.7 kB
system / Context	1 203	3	+1 200	66.8 kB	240 B	+66.6 kB

To better understand and diagnose performance issues such as I/O bottlenecks, long-running processes, CPU usage, and memory leaks, we can use clinic.js. This powerful tool provides a comprehensive suite of profiling features to analyze various aspects of a Node.js application.

```
Start your node server using clicnic.js by running
clinic doctor -- node server.js
```

It will generate an HTML report that contains a detailed analysis, including visual representations of the performance data such as CPU usage, memory consumption, and I/O operations. This report highlights areas of concern, like slow-running processes, memory leaks, or CPU bottlenecks, providing clear insights to help optimize and improve your application's performance.