title: Express.js Monitoring Integration description: Sematext Express.js monitoring integration is available as an npm package and required as any other Node.js module. Request rate and event loop latency, memory, http server stats, garbage collection and other Node.js reports and dashboards are available out of the box. Including HTTP logs and Application logs for API-specific troubleshooting and debugging.

This is the Express.js monitoring and logging agent for Sematext, written entirely in Node.js without CPU and memory overhead. It's easy to install and require in your source code.

Sematext Express.js Agent Quick Start

This lightweight, open-source Express.js monitoring agent collects Node.js process and performance metrics and sends them to Sematext. It is available as an npm package that can be added to Node.js source code like any other npm module.

First you install the npm module.

```
# Terminal
npm i sematext-agent-express
```

Configure Environment

Make sure to configure your ENVIRONMENT variables before adding `sematext-agent-express`. You can do this either by exporting the variables to your environment or by using `dotenv`. We suggest you use dotenv.

Export env vars

If you are using the US region of Sematext Cloud:

```
export REGION=US
export MONITORING_TOKEN=<YOUR_MONITORING_TOKEN>
export LOGS_TOKEN=<YOUR_LOGS_TOKEN>
export INFRA_TOKEN=<YOUR_INFRA_TOKEN>
```

Note: *The US region is set by default, so you do not need to add an environment variable at all.*

If you are using the EU region of Sematext Cloud:

```
export REGION=EU
export MONITORING_TOKEN=<YOUR_MONITORING_TOKEN>
export LOGS_TOKEN=<YOUR_LOGS_TOKEN>
export INFRA_TOKEN=<YOUR_INFRA_TOKEN>
```

Use dotenv

`npm i dotenv`

Create a `.env` file in the root of your project.

Add this code if you are using the US region of Sematext Cloud:

```
REGION=US
MONITORING_TOKEN=<YOUR_MONITORING_TOKEN>
LOGS_TOKEN=<YOUR_LOGS_TOKEN>
INFRA_TOKEN=<YOUR_INFRA_TOKEN>
```

Note: *The US region is set by default, so you do not need to add an environment variable at all.*

Add this code if you are using the EU region of Sematext Cloud:

```
REGION=EU
MONITORING_TOKEN=<YOUR_MONITORING_TOKEN>
LOGS_TOKEN=<YOUR_LOGS_TOKEN>
INFRA_TOKEN=<YOUR_INFRA_TOKEN>
```

Configure Agent

Make sure to load the environment variables at the top of your JavaScript entry point file. Then require `sematext-agent-express`.

The Agent has 3 parts:

- `stMonitor` - Monitors metrics and sends to Sematext Monitoring
- `stLogger` - A logger based on `winston`, that will send logs directly to Sematext Logs
- `stHttpLoggerMiddleware` - Express.js middleware function that will send all HTTP endpoint logs to Sematext Logs

Usage

```
// Load env vars
require('dotenv').config()

// require all agents
const {
  stMonitor,
  stLogger,
  stHttpLoggerMiddleware
} = require('sematext-agent-express')

// Start monitoring metrics
stMonitor.start()
```

```
// ...

// At the top of your routes add the stHttpLoggerMiddleware to send HTTP logs to Sematext
const express = require('express')
const app = express()
app.use(stHttpLoggerMiddleware)

// ...

// Use the stLogger to send all types of logs directly to Sematext
app.get('/api', (req, res, next) => {
  stLogger.info('Hello World.')
  stLogger.error('Some error.')
  res.status(200).send('Hello World.')
})
```

You can use all parts of the Agent of use them separately. It's all up to you.

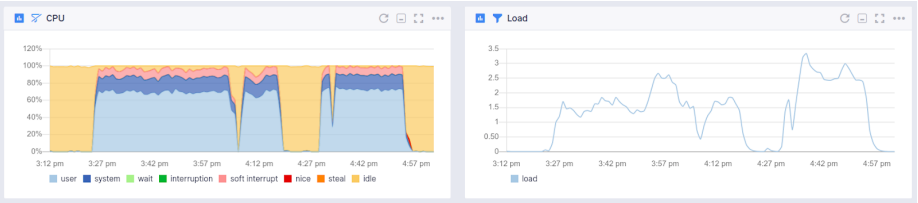
The Sematext Express.js Agent will start collecting dozens of key metrics and logs right away, and start showing you the performance and health of your Express.js applications immediately.

Collected Express.js Metrics

The Sematext Express.js Agent collects the following metrics.

Operating System

- CPU usage
- CPU load
- Memory usage



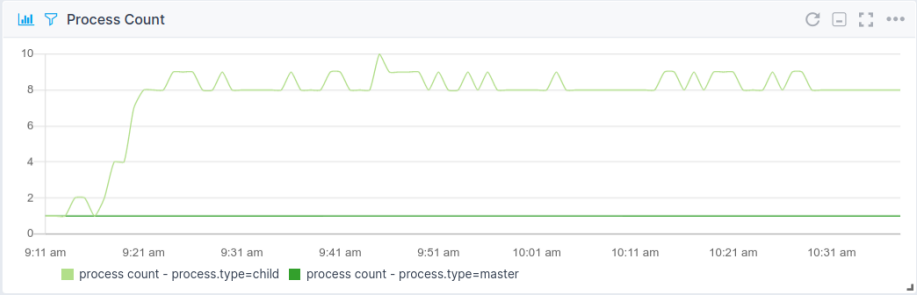
Process Memory Usage

- Released memory between garbage collection cycles
- Process heap size
- Process heap usage

Process Count

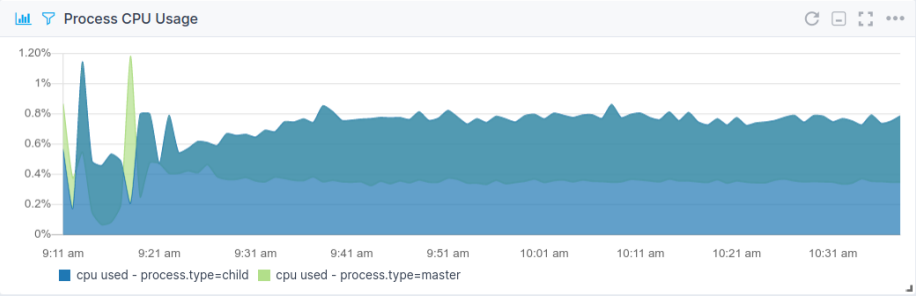
- Number of master processes

- Number of child processes



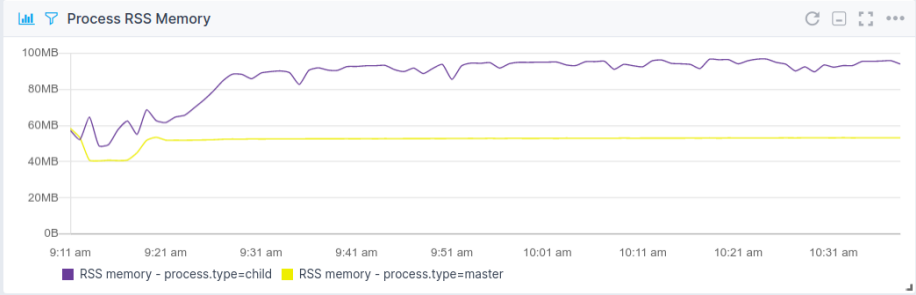
Process CPU Usage

- CPU usage per process
- CPU usage per PID



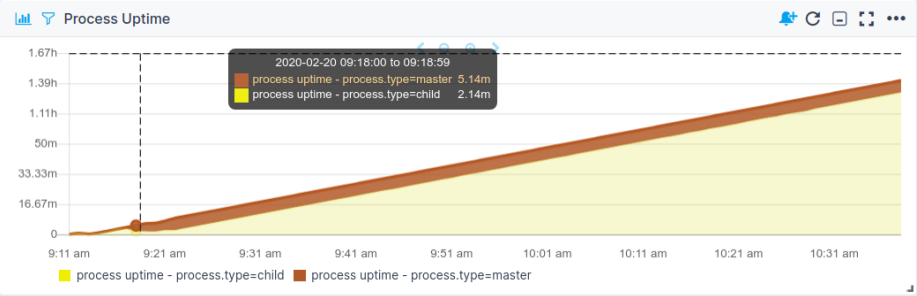
Process RSS Usage

- RSS usage per process
- RSS usage per PID



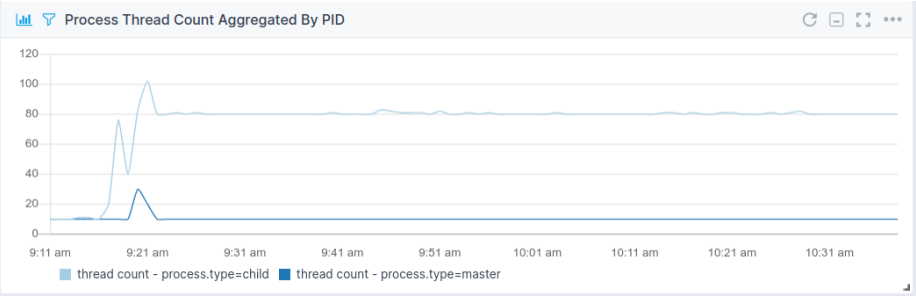
Process Uptime

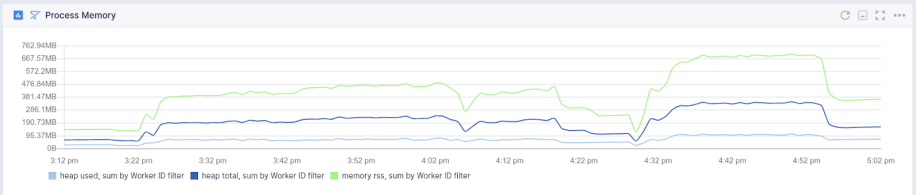
- Process Uptime per process
- Process Uptime per PID



Process Thread Count

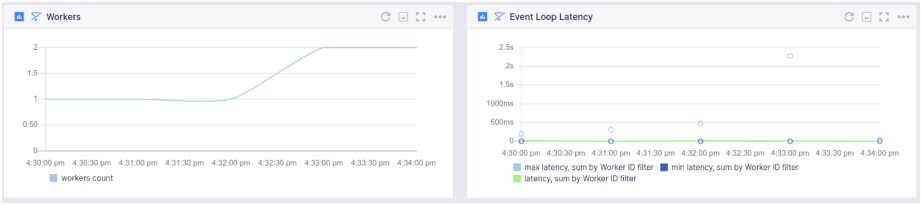
- Number of threads per process
- Number of threads per PID





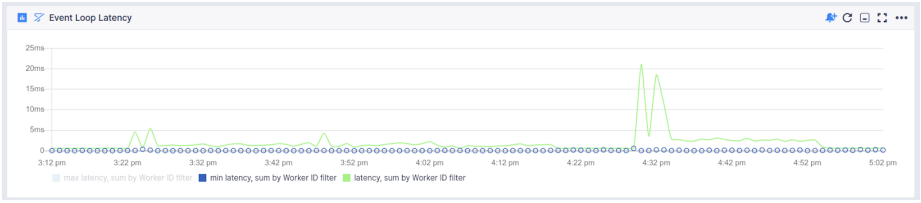
Worker Processes (cluster module)

- Worker count
- Event loop latency per worker



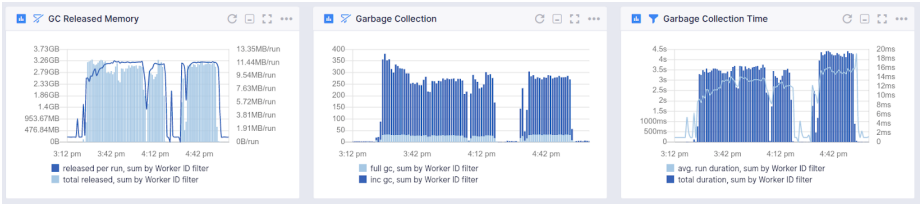
Event Loop

- Maximum event loop latency
- Minimum event loop latency
- Average event loop latency



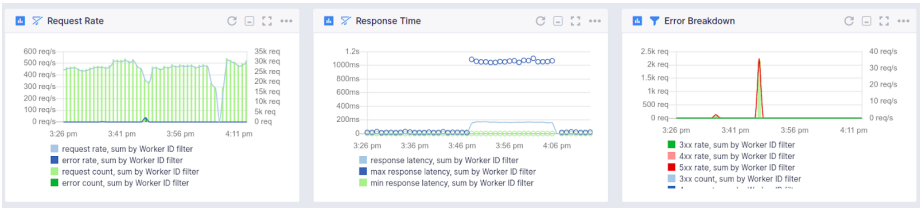
Garbage Collection

- Time consumed for garbage collection
- Counters for full garbage collection cycles
- Counters for incremental garbage collection cycles
- Released memory after garbage collection



HTTP Server Stats

- Request count
- Request rate
- Response time
- Request/Response content-length
- Error rates (total, 3xx, 4xx, 5xx)



Collected Express.js Logs

The Sematext Express.js Agent collects the following logs for every HTTP request.

Structured HTTP logs

Here's an example of what a structured HTTP log looks like:

timestamp	message	severity	host	ip	method	url	status	content length	response time
2019-12-23T21:06:36.615Z	your_message	info/debug/error	your_hostname	127.0.0.1	GET/PUT/POST/DELETE	/api/v1/xxx	2xx/3xx/4xx/5xx	bytes	ms

Displayed in `JSON` format:

```
{
  "@timestamp": "2019-12-23T21:06:36.615Z",
  "message": "HTTP LOG",
  "severity": "info",
  "host": "ip-172-31-29-198",
  "ip": "172.31.29.198",
  "method": "GET",
  "url": "/v1/users",
  "status": "200",
  "contentLength": "-",
  "responseTime": "18.403"
}
```

Using this data, you can create custom dashboards displaying detailed data about all HTTP requests hitting your Express.js application.

Running the Sematext Express.js Agent in Production

If you want to have a reliable application with high uptime, make sure to use the `cluster` module, with the addition of running the Node.js process with Systemd. This will ensure restarts when your application fails and make rolling

updates much easier.

Use the `cluster` module to run Node.js

To make use of the full power of your server, you should run an instance of your Node.js application on each CPU core. The `cluster` module makes this easier than ever. Create another file called `cluster.js`.

```
// cluster.js

const cluster = require('cluster')
const numCPUs = require('os').cpus().length
const app = require('./app')
const port = process.env.PORT || 3000

const masterProcess = () => Array.from(Array(numCPUs)).map(cluster.fork)
const childProcess = () => app.listen(port)
if (cluster.isMaster) masterProcess()
else childProcess()
cluster.on('exit', (worker) => cluster.fork())
```

Now you can run your app with:

```
node cluster.js
```

The cluster will spin up a master process with a dedicated process ID and run `numCPUs` number of worker processes. They will be load balanced in a round-robin fashion from the master process.

This is not all, you should also make sure to run your Node.js application with Systemd to make it a system service and run automatically on startup and restart itself if it fails.

Set up Node.js with Systemd

The service files for the things that systemd controls all live under the directory path

```
/lib/systemd/system
```

Create a new file there:

```
sudo vim /lib/systemd/system/app.service
```

And add this piece of code:

```
# /lib/systemd/system/app.service

[Unit]
Description=app.js - running your Node.js app as a system service
Documentation=https://yourwebsite.com
```

```
After=network.target

[Service]
Type=simple
User=root
ExecStart=/usr/bin/node /absolute/path/to/your/project/app.js
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

To use Systemd to control the app you first need to reload the Daemon to register the new file.

```
sudo systemctl daemon-reload
```

Now launch your app with:

```
sudo systemctl start app
```

You've successfully launched your Node.js app using Systemd! If it doesn't work for some reason, make sure to check your paths in `ExecStart` are correct.

```
ExecStart=/usr/bin/node /absolute/path/to/your/project/app.js
```

These need to point to the `node` binary and the absolute path to your `app.js` file.

Use `PM2` to run Node.js

You can also run your application with PM2 just like you would normally. Using the same setup as with a default Express.js server. Load the env vars and agent at the top of your source file.

```
// app.js

// Load env vars
require('dotenv').config({ path: '/absolute/path/to/your/project/.env' })

// require all agents
const {
  stMonitor,
  stLogger,
  stHttpLoggerMiddleware
} = require('sematext-agent-express')

// Start monitoring metrics
stMonitor.start()
```

```
// ...

// At the top of your routes add the stHttpLoggerMiddleware to send HTTP logs to Sematext
const express = require('express')
const app = express()
app.use(stHttpLoggerMiddleware)

// ...
```

Run the `pm2` command to start your server.

```
pm2 start app.js -i max
```

The agent will detect you are running PM2 and start collecting metrics automatically.

Integration

- Agent: https://github.com/sematext/sematext-express-agent
- Instructions: https://apps.sematext.com/ui/howto/Node.js/overview

Metrics

Metric Name	Key	Agg	Type	Description
heap total	nodejs.heap.size	Avg	Long	
heap used	nodejs.heap.used	Avg	Long	
total released	nodejs.gc.heap.diff	Sum	Double	
total duration	nodejs.gc.time	Sum	Double	
full gc	nodejs.gc.full	Sum	Long	
inc gc	nodejs.gc.inc	Sum	Long	
memory rss	nodejs.memory.rss	Avg	Long	
process count	process.count	All	Long	
process cpu usage	process.cpu.usage	All	Double	
process rss usage	process.rss	All	Double	
process thread count	process.thread.count	All	Long	
process uptime	process.uptime	All	Long	
workers count	nodejs.workers	Avg	Long	
request count	nodejs.requests	Sum	Long	
error count	nodejs.errors	Sum	Long	
5xx count	nodejs.errors.5xx	Sum	Long	
4xx count	nodejs.errors.4xx	Sum	Long	
3xx count	nodejs.errors.3xx	Sum	Long	
total req. size	nodejs.requests.size.total	Sum	Long	
total res. size	nodejs.response.size.total	Sum	Long	
min response latency	nodejs.responses.latency.min	Min	Long	
max response latency	nodejs.responses.latency.max	Max	Long	
min latency	nodejs.eventloop.latency.min	Min	Long	

Metric Name	Key	Agg	Type	Description
max latency	nodejs.eventloop.latency.max	Max	Long	

FAQ

How to Monitor OS and Infra metrics with the Node.js Integration?

We have deprecated the built-in Operating System monitor in the Node.js-based Agent and moved to using our Go-based Sematext Agent for Operating System and Infrastructure metrics. If you are using the `spm-agent-nodejs >=4.0.0` or the `sematext-agent-express >=2.0.0` you are required to install or upgrade the Sematext Agent to gather Operating System and Infrastructure metrics.

Can I install Express.js agent on Windows?

Yes. The native modules are automatically compiled during "npm install" (using node-gyp). On Windows the required build tools like python or C++ compilers are typically not installed by default. See https://github.com/TooTallNate/node-gyp for details about the required compiler and build tools.

How can I use Node.js agent behind Firewalls / Proxy servers?

By default data is transmitted via HTTPS. If no direct connection is possible, a proxy server can be used by setting the environment variable HTTPS_PROXY=https://your-proxy.

What should I do after upgrading to a new Node.js version?

If you switch the Node.js version the sematext-express-agent package will need to be installed again (due to the fact that included native modules may change from version to version). After the version change please run a fresh "npm install" if you added sematext-express-agent to the dependencies in your package.json or at the very least run "npm install sematext-express-agent".

How do I upgrade to the latest version of sematext-express-agent?

To use the latest version of sematext-express-agent we recommend you install/upgrade using:

```
npm install sematext-express-agent@latest
```

To add the dependency to your package.json simply use:

```
npm install sematext-express-agent@latest --save
```

Troubleshooting

If you are not seeing some or any Express.js metrics, you can create a "diagnostics dump" and contact us via live chat or email. To create the diagnostics dump just run the following command in the root directory of your project.

```
sudo node ./node_modules/spm-agent-nodejs/bin/spm-nodejs-diagnostics.js
```

This will create a ZIP file and show the Sematext Support email address to which the ZIP file should be sent.