Pitfalls and lessons learned with Node.JS

Juho Mäkinen
Software Architect at Applifier
@juhomakinen

Reaktor Dev Day
2011-09-02

Why would you choose Node.JS?

Using Javascript as a platform

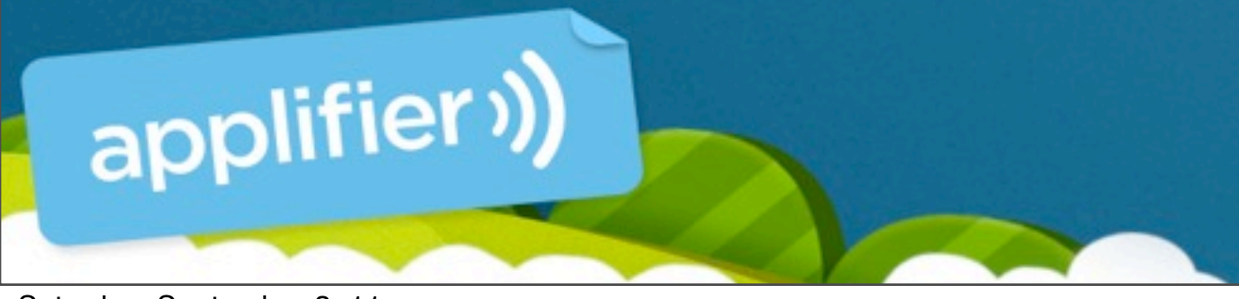
- It's possible to write entire application stack with Javascript
- From browser all the way to the datastore
- Imagine that you can send a JS object from the browser to the backend, store it to the database and retrieve it later without any unnecessary transformations.

Why Javascript

- Fast development cycle
- Easy data transformations as the storage and delivery is always JSON
- It's possible to share code between browser and the backend

Why Node.JS

- Perfect for low latency, high IO applications
  - Load balancing, API backend, comet / longpoll servers, streaming...
- It's easy to process the request, close the client socket and continue some tasks after the request has been processed.
- It's easy to cache data between requests



Where Node isn't good

- Harder to code (than say PHP).
- Need to take perfect care of all resources to avoid leaks.
- Less libraries and bindings because everything must be asynchronous and non-blocking.

General tips for Node.JS

What have we learned?

Buy a mac

- We can run our entire development environment in our mac laptops.
  - node
  - mongodb
  - apache, php (if you wish...)

Choose a good JS IDE

- I use PHPStorm from JetBrains
- Things to look:
  - Variable coloring by type
  - Unused code hilighting
  - Code navigation

Log everything

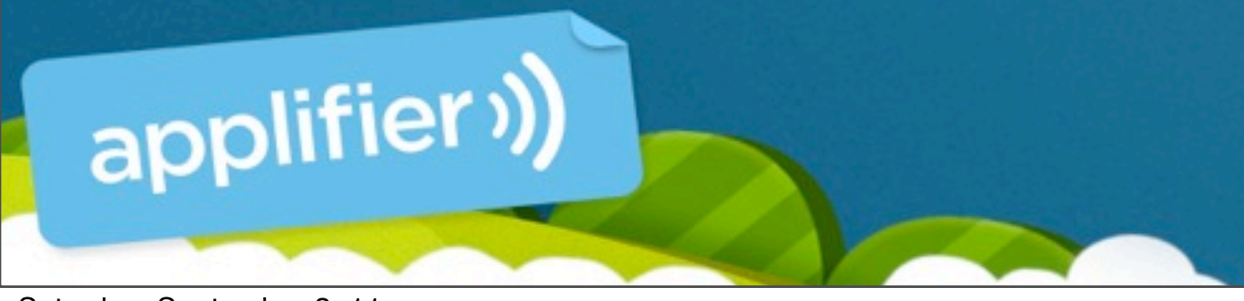
- Log all your data and store it somewhere when you can query it.
- Use some form of metrics system to trace counters and to get history graphs out of them.
- All this helps debugging, performance tuning and diagnostics.

Log everything

- We use Scribe to feed performance and log data from our machines into a hadoop cluster. https://github.com/facebook/scribe
  - Hadoop is used to crunch the data into usable metrics.
- There's also tools for getting realtime statistics into usable graphs.
  - Zabbix, statsd / graphite

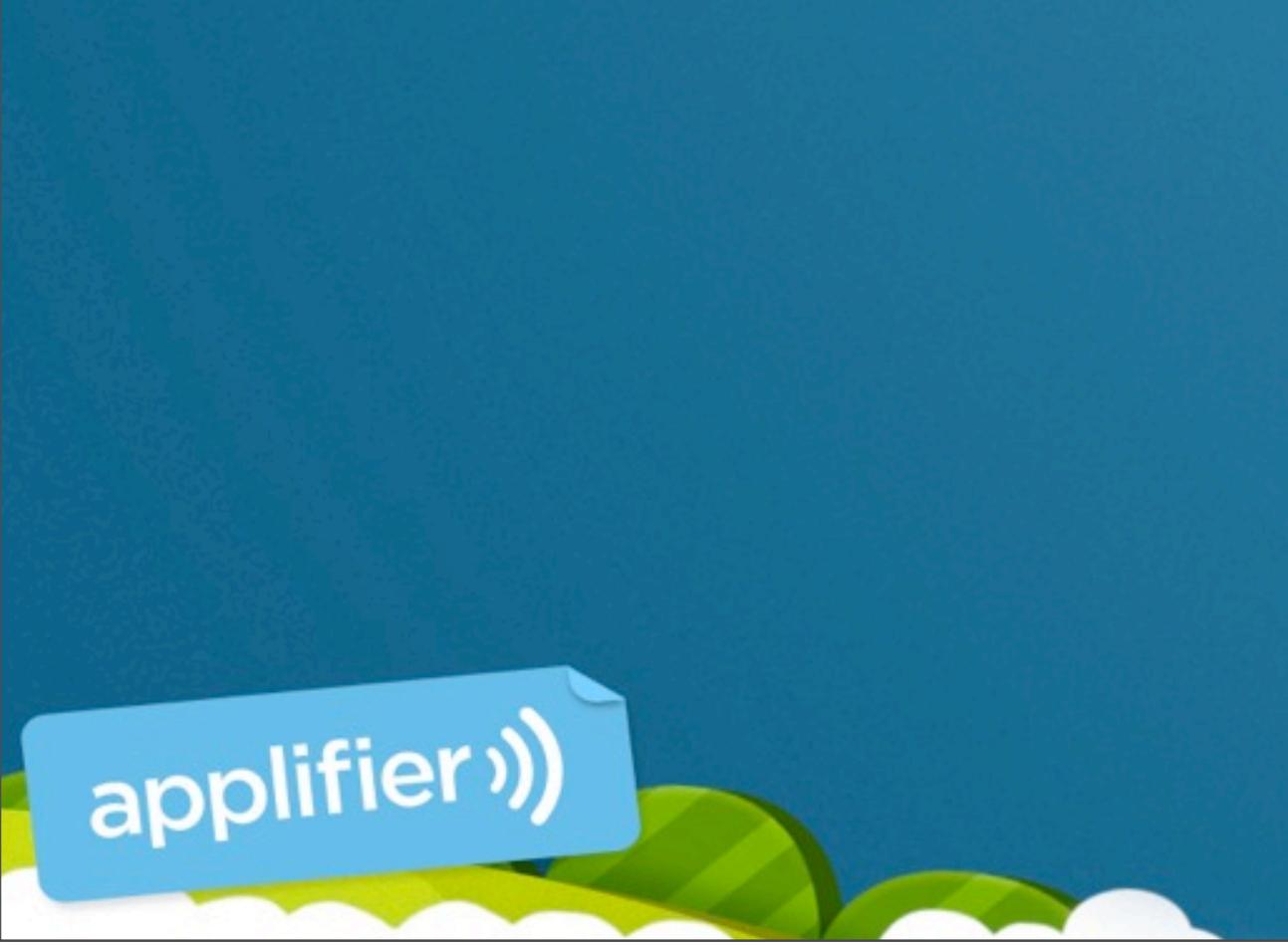
Clustering

- Node is single threaded
- Using multiple cores needs additional effort. There are two good options
  - Spawn multiple instances, one for each core and use load balancer to drive traffic into them. “Forever” makes this easy. https://github.com/indexzero/forever



Clustering

- Another option: Use Node Cluster (http://learnboost.github.com/cluster/) to fork worker threads.



Where to get modules?

- npm stands for “node package manager”
  - It’s your apt-get / CPAN / yum for all your node module needs. Learn it.
- Github. Full of node related modules.
- npm packages can be installed for each project, or globally for entire system.
  - Prefer to install packages per project.

Use async.js library

- Makes easier to write asynchronous code and makes it look prettier.
- Helps avoiding nesting callbacks
- https://github.com/caolan/async

Don’t deploy via NPM

- NPM packages gets upgrades
  - But you might not notice one
- An upgrade might break your system by introducing a bug in an npm module used by some other npm module.
- Hopefully your CI and testing environment catches this.

Watch out for leaks

- The node server stays intact between requests.
- Thus resources, memory and database connections can easily leak.
- Db connection leaks are most common.
- Our testing framework and database api helps detecting and avoiding them

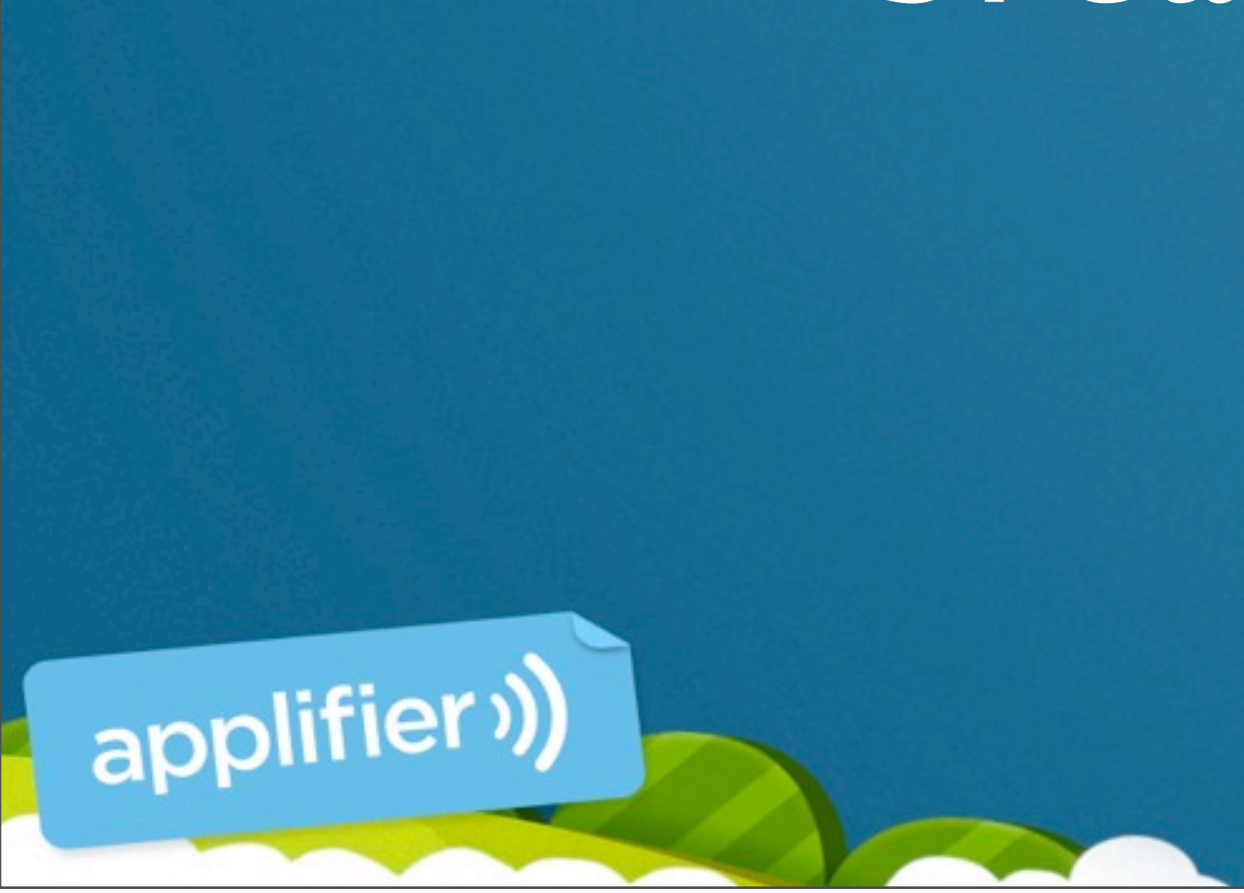
Detect socket leaks

- You can use “netstat -np” to display socket connections to the node process.
  - Run a bunch of requests to the node and then look if the database connections are closed correctly.

Detect resource leaks

- Leaked sockets, database connections or objects stored into scopes might leak memory.
- Run node with "--expose-gc --trace_gc" to get memory consumption data.
- Beware, the GC makes this hard to read
    - Run 100000 queries and see that the memory consumption is still about the same.

The fun part:
Making your code break less



Delete form array

```
var a = ['a', 'b', 'c', 'd'];

delete a[0];

console.dir(a);
```

['b', 'c', 'd']

Delete form array

console.log(a[0]) prints **undefined**
console.log(a[1]) prints **b**
console.log(a[2]) prints **c**
console.log(a[3]) prints **d**
console.length is **4**

happy bug hunting...

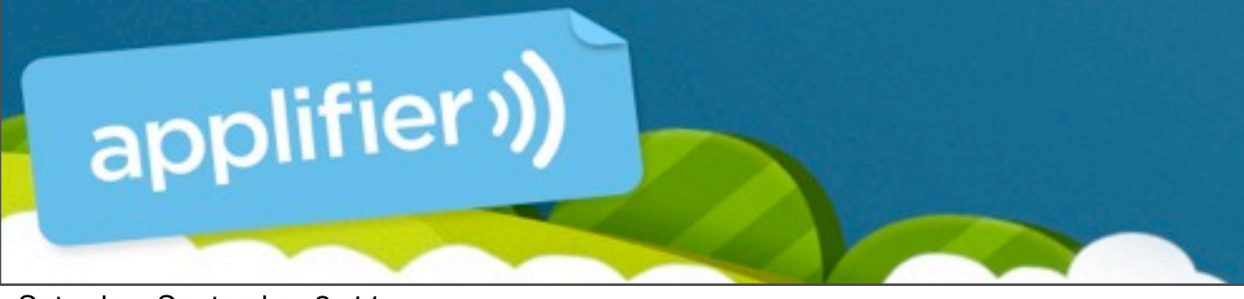
Delete form array

console.log(a[0]) prints **undefined**
console.log(a[1]) prints **b**
console.log(a[2]) prints **c**
console.log(a[3]) prints **d**
console.length is **4**

happy bug hunting...

Delete form array

```
// Array deletion done right
var a = ['a', 'b', 'c', 'd'];
// Array item 2 is ‘c’
// remove 1 item
a.splice(2, 1);
```



Delete form array

```
// Delete inside for loop
var a = ['a', 'b', 'b', 'c', 'd'];

for(var i = 0;i < a.length; i++) {
    if(a[i] == 'b') {
      a.splice(i, 1);
      i--;
    }
  }
```

Avoid try..catch blocks

- As node is event driven, you use a lot of callbacks.
- try..catch blocks doesn't behave well with callbacks and they're slow.
- Stick with the node callback notation where you pass error as first parameter and handle errors this way.

Don’t write callbacks unless needed

- Always prefer to return values directly with *return* instead of returning value in a callback.
- Also remember to refactor unnecessary callbacks away when a function turns from asynchronous into a synchronous after refactoring.

Separate sync and async gets with name

- *get* prefix for functions which returns something as value
  - var age = this.getAge();
- *fetch* prefix for functions which returns something via callback
  - this.fetchUser(function cb(err, user) {...});

Know your callstack

- The event callbacks come from the Node.JS main loop and this will ruin your callstack.

```
var level2 = function(msg) {
  console.log(new Error().stack);
};
var level1 = function(msg) { level2(msg); };

level2("Directly");
process.nextTick(function () { level2("nextTick"); });
```

Know your callstack

Directly:

at stack.js:5:15

at Object.<anonymous> (stack.js:12:1)

at Module._compile (module.js:404:26)

at Object..js (module.js:410:10)

at Module.load (module.js:336:31)

at Function._load (module.js:297:12)

at Array.0 (module.js:423:10)

at EventEmitter._tickCallback (node.js:126:26)

Know your callstack

Via process.nextTick()

at stack.js:5:15

at Array.0 (stack.js:13:32)

at EventEmitter._tickCallback (node.js:126:26)

Know your callstack

- You can use **long-stack-traces** npm module. Just require() it, but watch for small performance hit.

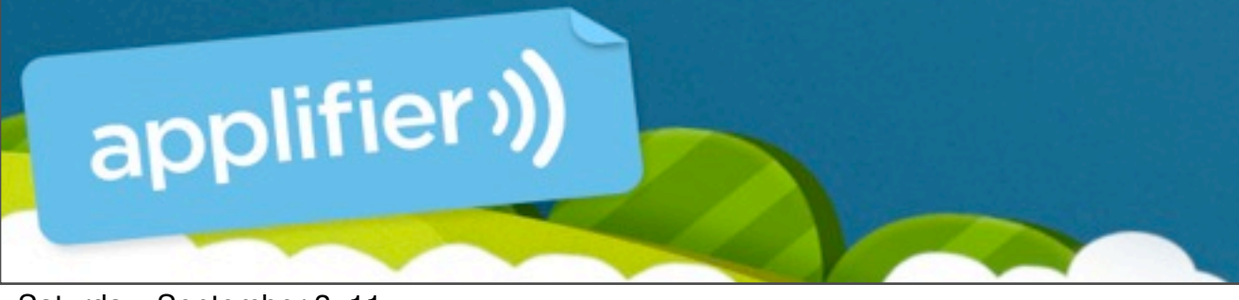
applifier

Avoind anonymous functions

- Give all your functions a name so that you can get better stacktraces.
- fs.rename(p1, p2, function **rename_cb**() {
  // callback name is rename_cb
  // instantly better stacktrace upon error
  }

Consider yielding

- Going over a long dataset might take too much time and delay other requests if you have real time requirements for the requests.
- You might want to yield using process.nextTick so that you give other requests change to complete.



Codepath complexity

- You can accidentally create a complex code path which is traversed multiple times in an error condition.

```
async.parallel([
  function func1(cb) { .... cb(); },
  function func2(cb) { .... cb(); }
], function all_done(err) {
  console.log("Both functions done");
});
```

Codepath complexity

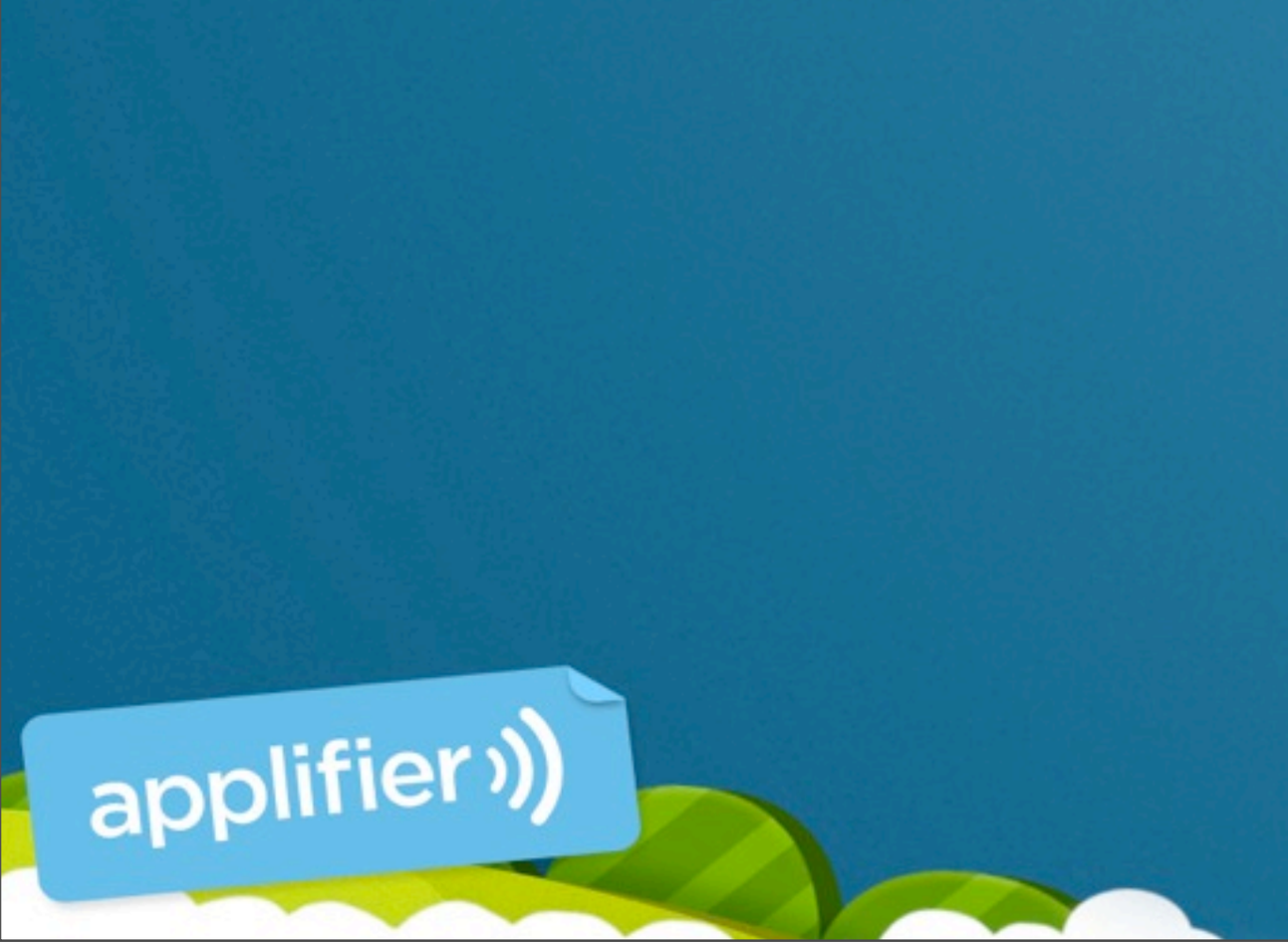
- Now think that both func1 and func2 start a complex IO operation.
  - func1 encounters an error and calls cb(“error”) early.
  - all_done() is now called with the error message.

Codepath complexity

- the func2 execution is not terminated on this error and thus it will continue.
- You need to design these kind of code paths so that they don't do any harm if some of the concurrent code paths will encounter an error.

Codepath complexity

- Add some try..catch blocks and you'll find yourself in a very serious and deep sh^H^H hole.



Questions?

Twitter: @juhomakinen