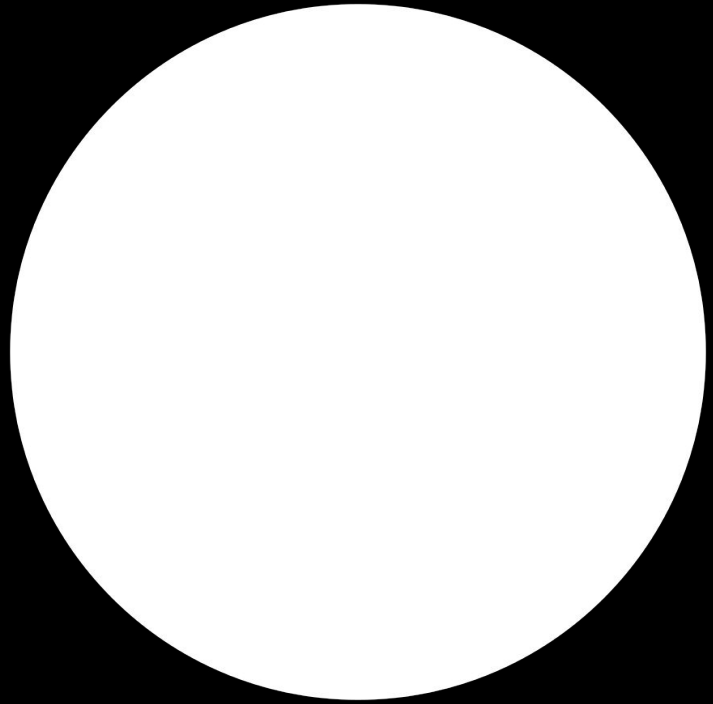


Metarhia

Metarhia

Community and Technology Stack
for Distributed Highload Applications
and Data Storage

Timur Shemsedinov
@tshemsedinov

Alexey Orlenko
@aqrln

- Open Lectures (>1000 stud/year), 75 lect/semester
- Meetups
  - meetup.com/KievNodeJS (1349) 19 since 2014
  - meetup.com/NodeUA (1679) 13
  - meetup.com/HowProgrammingWorks (1206) 54
- Channels
  - t.me/metarhia (640)
  - t.me/HowProgrammingWorks (865)
- Groups
  - t.me/nodeua (358)
  - t.me/MetarhiaHPW (352)

- Conceptual code
  - It’s short and impressive parts, I’ll show you
  - conceptual code becomes libs and examples
- Examples
  - github.com/HowProgrammingWorks
  - github.com/HowProgrammingWorks/Dictionary
  - contributors: 36, repos: 87, >500 examples
- Open Source
  - Technology stack: github.com/metarhia
- Contribution
  - Node.js and other projects contribution


四川电视节
Sichuan TV Festival


THE
SEVENTH
SENSE
WHAT MAY OCCUR AT
IN DIFFERENT 3 PLAC


计 算 机 学 院
COMPUTER SCHOOL

- Impress Application Server
  for highload clusters and private clouds with Node.js
- JSTP protocol: RPC, event bus, db sync on TCP, TLS, WebSocket and JSON5; with SDK for mobile and desktop: JavaScript, Java, Swift, C++, Haskell, Python, Objective-C, PHP, Golang, C#
- GlobalStorage: distributed reactive in-memory DBMS, work in progress
- Metasync: asynchronous programming abstractions
- Maojian: testing framework

- Impress Application Server
  https://github.com/metarhia/impress
- JSTP protocol
  https://github.com/metarhia/jstp
- GlobalStorage
  https://github.com/metarhia/globalstorage
- Metasync
  https://github.com/metarhia/metasync
- Maojian
  https://github.com/metarhia/maojian

```
gs.select({ category: 'Person', name: 'Marcus' })
  .filter(person => person.isImperor())
  .projection({
    name: ['name', toUpperCase],
    age: ['birth', toAge],
    place: ['address', getCity, getLatLon],
  })
  .order('age')
  .on('update', data => api.log(data.toString()))
  .clone().group('age')
  .map(x => x.count + ' imperors of age ' + x.age)
  .pipe(destinationCursor)
  .fetch((err, data) => Maybe(data)
    .map(api.jstp.emit('imperorsReportUpdated', data))
  )
  .clone().count().fetch((err, count) => Maybe(count)
    .map(api.log('Recalculated records: ' + count))
  );
```

```
const projection = (meta, obj) => (
  Object.keys(meta).reduce((hash, key) => (
    hash[key] = meta[key].reduce(
      (val, fn, i) => (i === 0 ? obj[fn] : fn(val)), null
    ), hash
  ), {})
);

const persons = [
  { name: 'Marcus Aurelius', city: 'Rome', born: 121 },
  { name: 'Victor Glushkov', city: 'Rostov on Don', born: 1923 },
  { name: 'Ibn Arabi', city: 'Murcia', born: 1165 },
  { name: 'Mao Zedong', city: 'Shaoshan', born: 1893 },
  { name: 'Rene Descartes', city: 'La Haye en Touraine', born: 1596 }
];

const md = {
  name: ['name'],
  place: ['city', upper, s => '<' + s + '>'],
  age: ['born', age]
};

const p1 = curry(projection)(md);
const data = persons.map(p1);
console.dir(data);
```

```
const projection = (meta) => {
  const keys = Object.keys(meta);
  return obj => {
    const hash = {};
    let name, key, def, val, fn;
    for (key of keys) {
      def = meta[key];
      [name, fn] = def;
      val = obj[name];
      if (fn) val = fn(val);
      hash[key] = val;
    }
    return hash;
  };
};
```

JavaScript Functional Example

```
const densityCol = 3;
const cellWidth = [18, 10, 8, 8, 18, 6];

const renderTable = table => table
  .map(row => row
    .map((cell, i) => cell
      .toString()[(i ? 'padStart' : 'padEnd')](cellWidth[i]))
    .join(''))
  .join('\n');

const proportion = (max, val) => Math.round(val * 100 / max);

const calcProportion = table => (
  table.sort((row1, row2) => (row2[densityCol] - row1[densityCol])),
  table.map(row => (
    row.push(proportion(table[0][densityCol], row[densityCol])), row
  ))
);

const getDataset = file => fs.readFileSync(file).toString()
  .split('\n')
  .filter((s, i) => i && s)
  .map(line => line.split(','));

const main = compose(getDataset, calcProportion, renderTable);
```

JS-friendly FP with chaining code alignment
With constants (not only functional expressions)
and not FP-friendly: Array.prototype.sort, .push

JavaScript Functional Example

```
const cellPad = (i, s, width) => (i ? s.padStart(width) : s.padEnd(width));
const cellWidth = i => [18, 10, 8, 8, 18, 6][i];

const renderCell = (cell, i) => cellPad(i, cell + '', cellWidth(i));
const renderRow = row => row.map(renderCell).join('');
const renderTable = table => table.map(renderRow).join('\n');

const densityCol = () => 3;
const proportion = (max, val) => Math.round(val * 100 / max);
const sortByDensity = table => table.sort(
  (row1, row2) => (row2[densityCol()] - row1[densityCol()])
);
const calcColumn = (table, max) => table.map(
  row => (row.push(proportion(max, row[densityCol()])), row)
);
const calcProportion = table => calcColumn(table, table[0][densityCol()]);

const parseTable = lines => lines.map(line => line.split(','));
const toLines = data => data.split('\n').filter((s, i) => i && s);
const readFile = file => fs.readFileSync(file).toString();
const getDataset = compose(readFile, toLines, parseTable);

const main = compose(getDataset, sortByDensity, calcProportion, renderTable);
```

Almost pure FP but excessively decomposed.
The whole meaning is lost behind the details,
divided into many small functions.

Not FP-friendly: Array.prototype.sort, .push
Not JS-friendly: lambda-constant, (...expr, return)

JavaScript Functional Example

Not a pure FP but quite readable and js-friendly

```
const DENSITY_COL = 3;

const renderTable = (table) => {
  const cellWidth = [18, 10, 8, 8, 18, 6];
  return table.map(row => (
    row.map((cell, i) => {
      const width = cellWidth[i];
      return i ? cell.toString().padStart(width) : cell.padEnd(width);
    }).join('')
  )).join('\n');
};

const proportion = (max, val) => Math.round(val * 100 / max);

const calcProportion = (table) => {
  table.sort((row1, row2) => row2[DENSITY_COL] - row1[DENSITY_COL]);
  const maxDensity = table[0][DENSITY_COL];
  table.forEach(row => {
    row.push(proportion(maxDensity, row[DENSITY_COL]));
  });
  return table;
};

const getDataset = (file) => {
  const lines = fs.readFileSync(file).toString().split('\n');
  lines.shift();
  lines.pop();
  return lines.map(line => line.split(','));
};

const main = compose(getDataset, calcProportion, renderTable);
```

```
// EventEmitter

const emitter = (events = {}) => ({
  on: (name, fn) => (events[name] = events[name] || []).push(fn),
  emit: (name, ...data) => (events[name] || []).forEach(fn => fn(...data))
});


const emitter = (l, o) => (l = {}, o = {
  on: (n, f) => (l[n] = l[n] || []).push(f),
  emit: (n, ...d) => (l[n] || []).map(f => f(...d)),
  once: (n, f, g) => o.on(n, g = (...a) => (f(...a), o.remove(n, g))),
  remove: (n, f, e) => (e = l[n] || [], e.splice(e.indexOf(f), 1)),
  clear: (n) => (n ? delete l[n] : l = {}),
  count: (n) => (l[n] || []).length,
  listeners: (n) => (l[n] || []).slice(),
  names: () => Object.keys(l)
});
```

Make async pool from factory

```
const poolify = (factory, min, norm, max) => {
  const pool = (par) => {
    if (typeof(par) !== 'function') {
      if (pool.items.length < max) {
        const delayed = pool.delayed.pop();
        if (delayed) delayed(par);
        else pool.items.push(par);
      }
      return;
    }
    if (pool.items.length < min) {
      const grow = norm - pool.items.length;
      const items = duplicate(factory, grow);
      pool.items.push(...items);
    }
    const res = pool.items.pop();
    if (res) par(res);
    else pool.delayed.push(par);
  };
  return Object.assign(pool, {
    items: duplicate(factory, norm),
    delayed: []
  });
};
```

```
// Usage
// 4kb buffer factory
const buffer = () => new Uint32Array(128);
// Factory to pool
const pool = metasync.poolify(buffer, 10, 15, 20);
// Async get from pool
pool(item1 => {
  pool(item2 => {
    // Return to pool
    pool(item1);
    pool(item2);
  });
});
// Async test
const next = () => {
  pool(item => {
    i++;
    if (i < 10) {
      setTimeout(next, i * 10);
      setTimeout(() => pool(item), i * 100);
    }
  });
};
next();
```

poolify(factory, min, norm, max)
- factory - pass a function to instantiate pool objects
- min - low watermark of pool instances
- norm - default amount of pool instances
- max - high watermark of pool instances
- returns pool function
  - pool(instance => {}) to get instance or pool(instance) to recycle

Быстрый функтор на прототипах и замыканиях

```
function Counter() {}

const counter = initial => {
  const f = val => {
    f.count += val;
    Object.keys(f.events).filter(n => n <= f.count).forEach(n => {
      f.events[n].forEach(callback => callback(f.count));
      delete f.events[n];
    });
    return f;
  };
  Object.setPrototypeOf(f, Counter.prototype);
  return Object.assign(f, { count: 0, events: {} })(initial);
};

Counter.prototype.on = function(n, callback) {
  const event = this.events[n];
  if (event) event.push(callback); else this.events[n] = [callback];
  return this(0);
};
```

Problems of Software Engineering

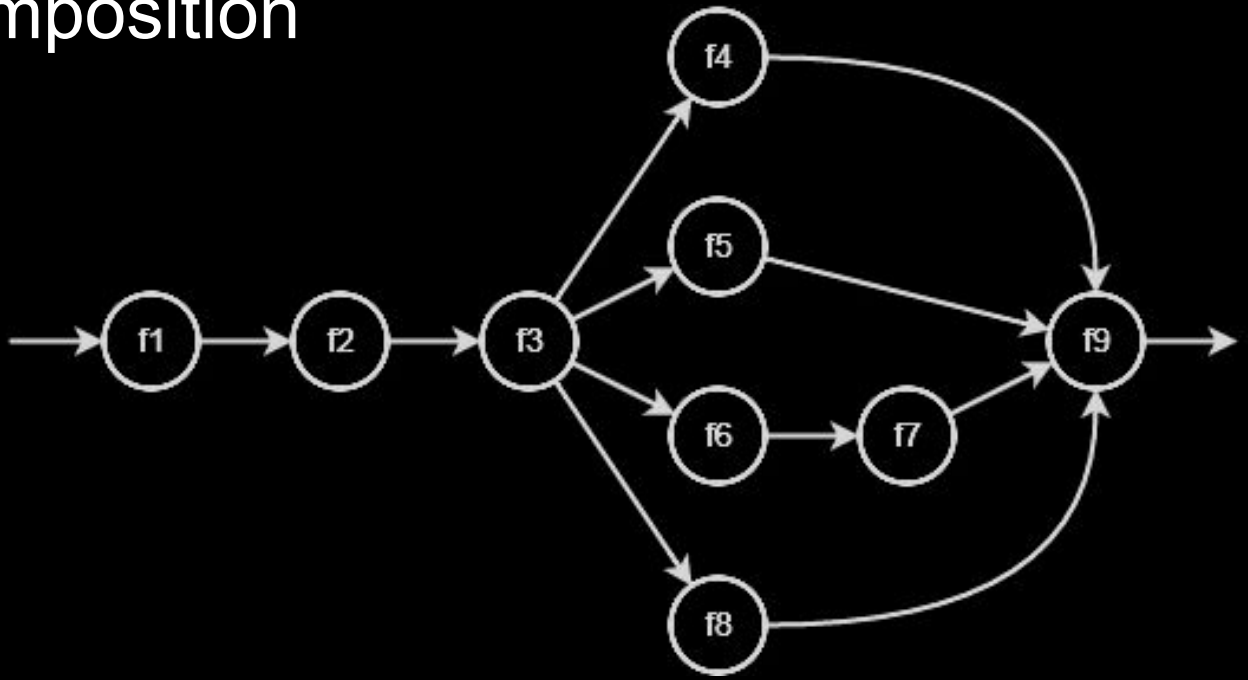
- Not scientific decision-making
- Voluntarism, no answers
- Fundamental knowledge is not popular
- Quality is not a goal for business
- No sense in work, coded for trash

Functor + Chaining + composition

```
const c1 = chain()
  .do(readConfig, 'myConfig')
  .do(doQuery, 'select * from cities')
  .do(httpGet, 'http://kpi.ua')
  .do(readFile, 'README.md');

c1();
```

```
function chain(prev = null) {
  const cur = () => {
    if (cur.prev) {
      cur.prev.next = cur;
      cur.prev();
    } else {
      cur.forward();
    }
  };
  cur.prev = prev;
  cur.fn = null;
  cur.args = null;
  cur.do = (fn, ...args) => {
    cur.fn = fn;
    cur.args = args;
    return chain(cur);
  };
  cur.forward = () => {
    if (cur.fn) {
      cur.fn(cur.args, () => {
        if (cur.next) {
          cur.next.forward();
        }
      });
    }
  };
  return cur;
}
```

Asynchronous composition




```
const fx = metasync(
  [f1, f2, f3, [[f4, f5, [f6, f7], f8]], f9]
);
```

Concurrent Queue

```
const cq = metasync.queue(3)
  .wait(2000)
  .timeout(5000)
  .throttle(100, 1000)
  .process((item, cb) => cb(err, result))
  .success((item) => {})
  .failure((item) => {})
  .done(() => {})
  .drain(() => {});
```

Collector

```
const dc1 = metasync
  .collect(3)
  .timeout(5000)
  .done((err, data) => {});
dc1(item);

const dc2 = metasync
  .collect(['key1', 'key2', 'key3'])
  .timeout(5000)
  .done((err, data) => {});
dc2(key, value);
```

- Websocket Chat

  https://github.com/HowProgrammingWorks/WebsocketChat
- LiveTable (interactive, multiuser)

  https://github.com/HowProgrammingWorks/LiveTable
- Abstraction Layers

  https://github.com/HowProgrammingWorks/AbstractionLayers
- IoC and DI

  https://github.com/HowProgrammingWorks/InversionOfControl
  https://github.com/HowProgrammingWorks/DependencyInjection
- Transactions

  https://github.com/HowProgrammingWorks/Transaction

Metarhia Technology Stack Key Ideas

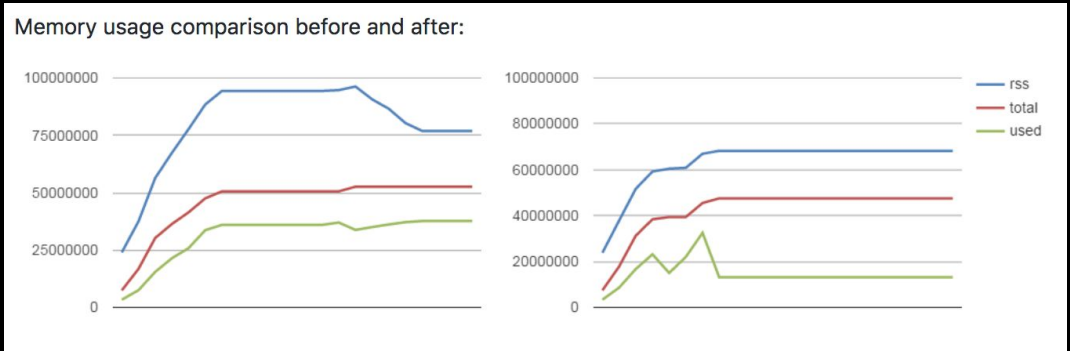
- Unification: API, data, contracts
- Homogeneity of server infrastructure
- No back compatibility
- Open Source & no vendor lock
- Architectural decisions
- Community and trainings

Node.js

- 3 Node.js contributors
  @tshemsedinov, @aqrln, @belochub
- 1 Node.js core collaborator
  @aqrln
- Notable contributions:
  keepAliveTimeout for HTTP, substantially faster Unix domain sockets, faster base64 decoding, pushing N-API adoption forward, community work

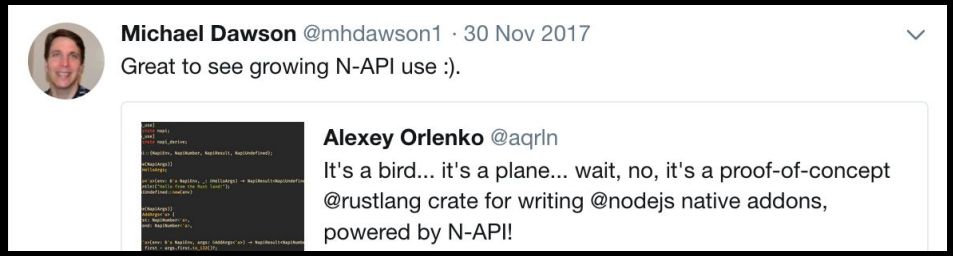
Node.js

- https://habrahabr.ru/post/264851/
- https://aqrln.github.io/kyivjs-2017




```
Comparing before, beforeBench.txt ( 37 lines) to after, afterBench.txt ( 37 lines)
clients: 1:         0 ->       0 ops/sec (Δ -       0 -     NaN%)
PING:           27089 ->   30315 ops/sec (Δ +    3226 +   11.91%)
PING:          618513 ->  668113 ops/sec (Δ +   49600 +    8.02%)
SET 4B str:     25407 ->   26432 ops/sec (Δ +    1025 +    4.03%)
SET 4B str:    432767 ->  446921 ops/sec (Δ +   14154 +    3.27%)
SET 4B buf:     12026 ->   12261 ops/sec (Δ +     235 +    1.95%)
SET 4B buf:     17086 ->  171152 ops/sec (Δ + 154066 + 901.71%)
GET 4B str:     28040 ->   28977 ops/sec (Δ +     937 +    3.34%)
GET 4B str:    507737 ->  525030 ops/sec (Δ +   17293 +    3.41%)
GET 4B buf:     23984 ->   27532 ops/sec (Δ +    3548 +   14.79%)
GET 4B buf:    454798 ->  523850 ops/sec (Δ +   69052 +   15.18%)
SET 4KiB str:   20527 ->   23892 ops/sec (Δ +    3365 +   16.39%)
SET 4KiB str: 140304 ->  154618 ops/sec (Δ +   14314 +   10.20%)
SET 4KiB buf:   10802 ->   11967 ops/sec (Δ +    1165 +   10.79%)
SET 4KiB buf:   15961 ->  129128 ops/sec (Δ + 113167 + 709.02%)
GET 4KiB str:   23923 ->   25663 ops/sec (Δ +    1740 +    7.27%)
GET 4KiB str: 171351 ->  187105 ops/sec (Δ +   15754 +    9.19%)
GET 4KiB buf:   23174 ->   26569 ops/sec (Δ +    3395 +   14.65%)
GET 4KiB buf: 144582 ->  163794 ops/sec (Δ +   19212 +   13.29%)
INCR:           25148 ->   27120 ops/sec (Δ +    1972 +    7.84%)
INCR:          471591 ->  534386 ops/sec (Δ +   62795 +   13.32%)
LPUSH:          24301 ->   26746 ops/sec (Δ +    2445 +   10.06%)
LPUSH:         383287 ->  441743 ops/sec (Δ +   58456 +   15.25%)
LRANGE 10:      23138 ->   25018 ops/sec (Δ +    1880 +    8.13%)
LRANGE 10:     204398 ->  236106 ops/sec (Δ +   31708 +   15.51%)
LRANGE 100:     15573 ->   17351 ops/sec (Δ +    1778 +   11.42%)
LRANGE 100:     33627 ->   37870 ops/sec (Δ +    4243 +   12.62%)
SET 4MiB str:     196 ->     215 ops/sec (Δ +      19 +    9.69%)
SET 4MiB str:     193 ->     221 ops/sec (Δ +      28 +   14.51%)
SET 4MiB buf:     398 ->     461 ops/sec (Δ +      63 +   15.83%)
SET 4MiB buf:     415 ->     468 ops/sec (Δ +      53 +   12.77%)
GET 4MiB str:     415 ->     466 ops/sec (Δ +      51 +   12.29%)
GET 4MiB str:     211 ->     227 ops/sec (Δ +      16 +    7.58%)
GET 4MiB buf:     295 ->     321 ops/sec (Δ +      26 +    8.81%)
GET 4MiB buf:     296 ->     318 ops/sec (Δ +      22 +    7.43%)
:               85250 ->   85161 ops/sec (Δ -      89 -    0.10%)
Mean difference in ops/sec:  +18075.4
```

N-API (Node.js API) for Rust



M E M bI

Alexey Orlenko
@aqrln

no more imperative javascript in 2018, javascript is a fancy lisp flavor from now on

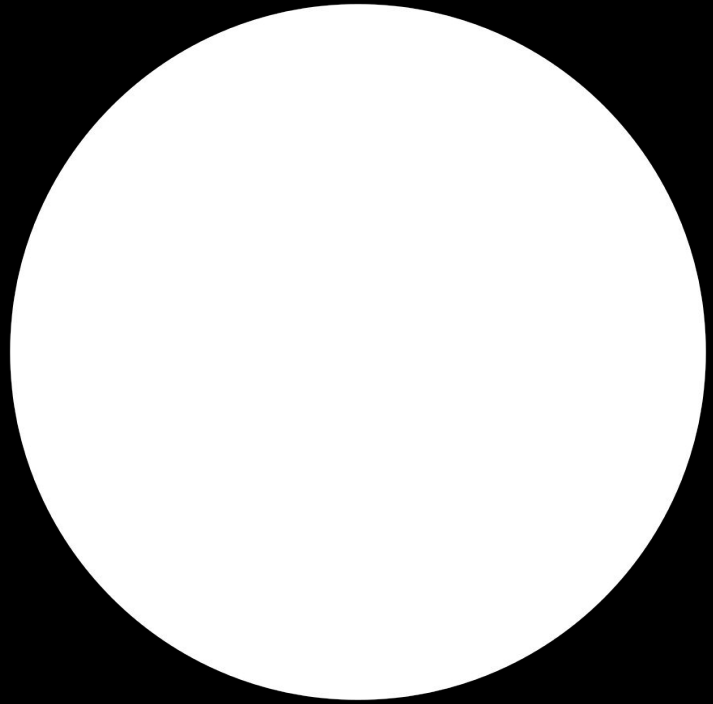
```
fmap = (f ⇒ g ⇒ x ⇒ (f (g (x))))

curryOnce = (f ⇒ x ⇒ ( ...xs) ⇒ (f (x, ...xs)))
curry = (n ⇒ f ⇒ (
  ((n == 1) ?
    (f) :
    (fmap (curry (n - 1)) (curryOnce (f))))))

method = (m ⇒ (m.call.bind (m)))
methodWithArgs = (n ⇒ m ⇒ (curry (n) (method (m))))
flip = (f ⇒ x ⇒ y ⇒ (f (y) (x)))

split = (methodWithArgs (2) (''.split))
map = (flip (methodWithArgs (2) ([].map)))

;;;

(console.log
  (map (x ⇒ (x * 2))
       (split ('1 2 3 4 5') (' '))))
~
~
NORMAL ./curry.js                                javascript.jsx JS
"./curry.js" 20L, 465C written

➜ curry node curry.js
[ 2, 4, 6, 8, 10 ]
➜ curry
```

4:45 AM - 19 Jan 2018

357 Retweets **1,033** Likes

Timur Shemsedinov
tshemsedinov@github
timur.shemsedinov@gmail.com
tshemsedinov@facebook
marcusaurelius@habrahabr

Alexey Orlenko
aqrln@github
aqrln@twitter
eaglexrlnk@gmail.com



Metarhia