REDISCONF 2016

BACKGROUND TASKS IN NODE.JS

A Survey using Redis

@EVANTAHLER

WARNING

MOST OF THE IDEAS IN THIS PRESENTATION ARE ACTUALLY VERY BAD IDEAS.

TRY THESE AT ~, NOT ON PRODUCTION

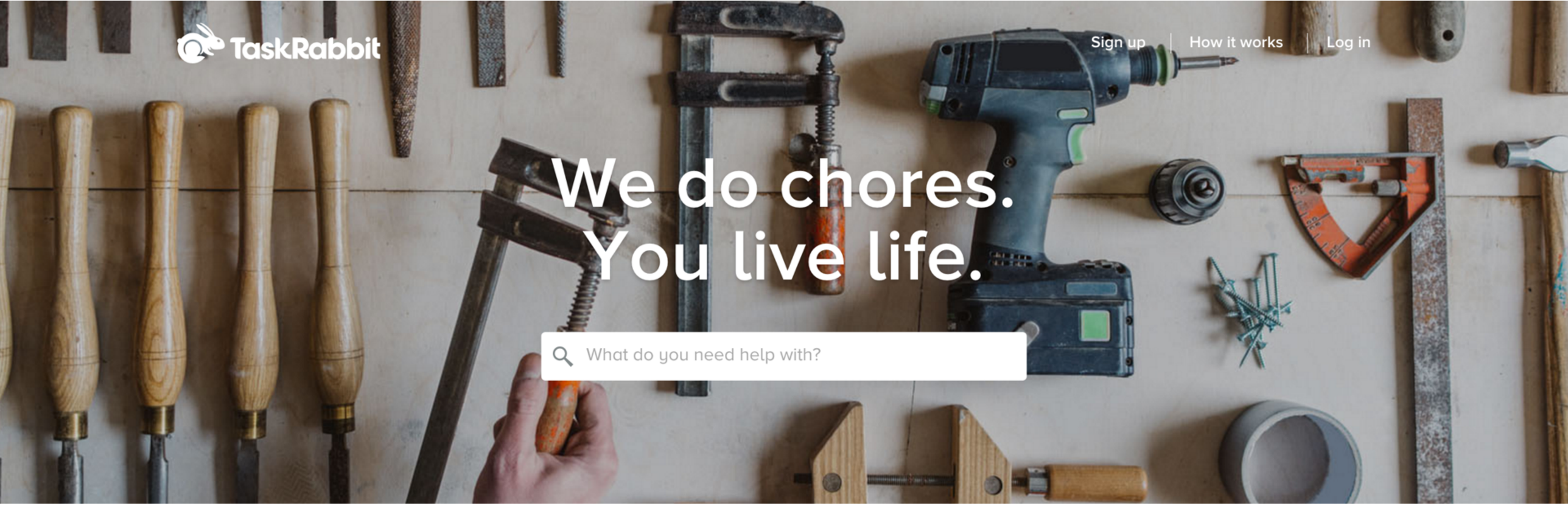

node
JS

HI. I'M EVAN

- Director of Technology @ TaskRabbit
- Founder of ActionHero.js, node.js framework
- Node-Resque Author

@EVANTAHLER

BLOG.EVANTAHLER.COM



How it Works

Pick a Task

Choose from a list of popular chores and errands

Get Matched

We'll connect you with a skilled Tasker within minutes of your request

Get it Done

Your Tasker arrives, completes the job and bills directly in the app

WHAT IS NODE.JS

- Server-side Framework, uses JS
- Async
- Fast

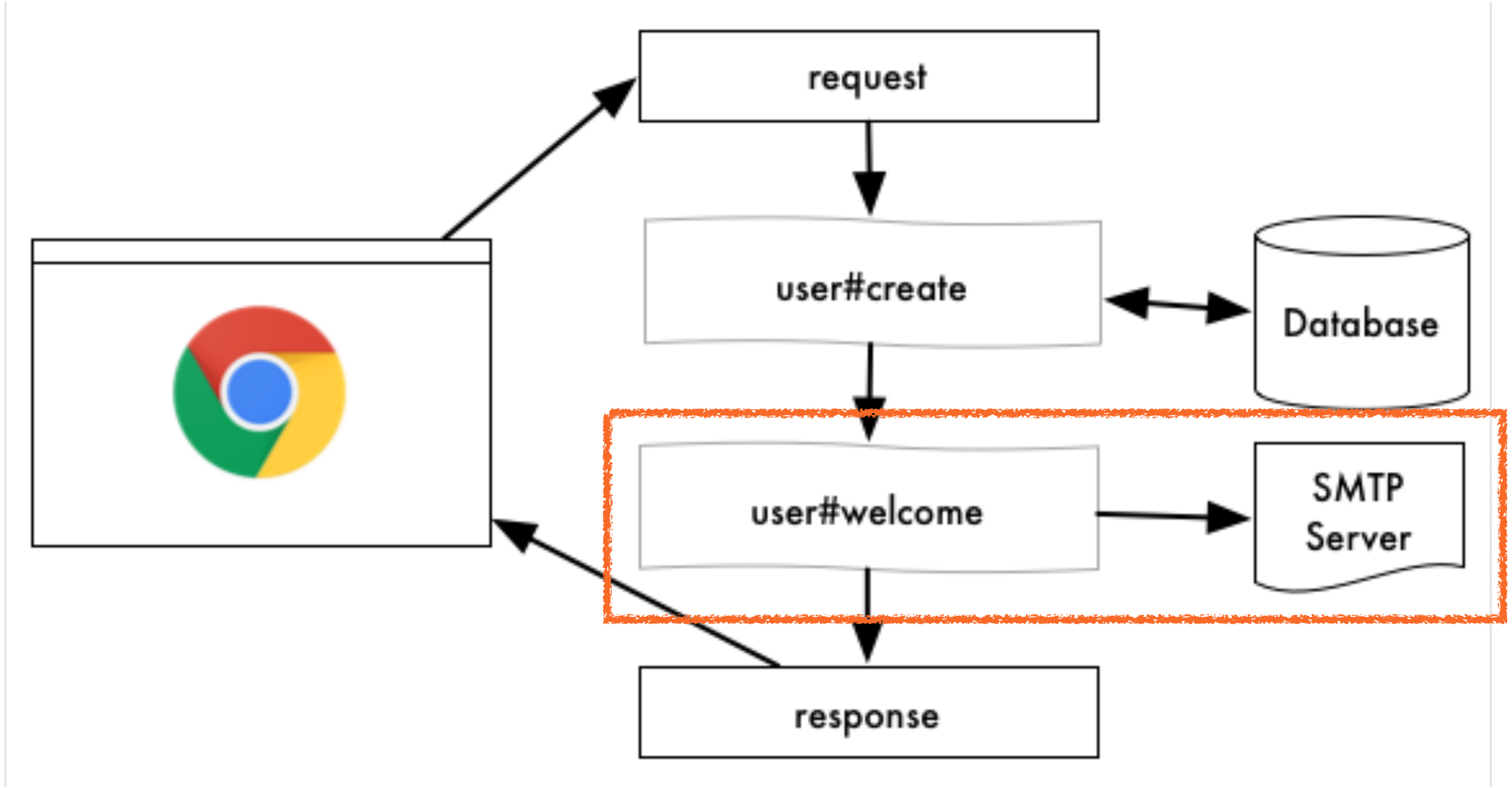
WHAT IS REDIS

- In-memory database
- Structured data
- Fast

EVERYTHING IS FASTER IN NODE...
ESPECIALLY THE BAD IDEAS

Me (Evan)

WHAT ARE WE GOING TO TALK ABOUT?



POSSIBLE TASK STRATEGIES:

FOREGROUND (IN-LINE)

PARALLEL (THREAD-ISH)

LOCAL MESSAGES (FORK-ISH)

REMOTE MESSAGES (*MQ-ISH)

REMOTE QUEUE (REDIS + RESQUE)

IMMUTABLE EVENT BUS (KAFKA-ISH)

1) FOREGROUND TASKS

php Downloads Documentation Get Involved Help

```php
<?php
$to      = 'nobody@example.com';
$subject = 'the subject';
$message = 'hello';
$headers = 'From: webmaster@example.com' . "\r\n" .
    'Reply-To: webmaster@example.com' . "\r\n" .
    'X-Mailer: PHP/' . phpversion();

mail($to, $subject, $message, $headers);
?>
```

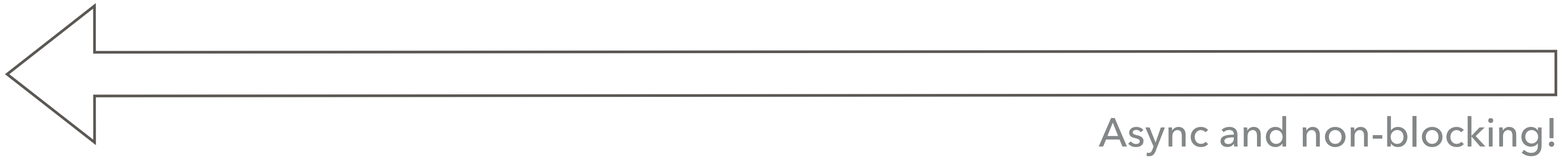
SENDING EMAILS

```
var http          = require('http');
var nodemailer    = require('nodemailer');
var httpPort      = process.env.PORT || 8080;
var httpHost      = process.env.HOST || '127.0.0.1';

var transporter = nodemailer.createTransport({
    service: 'gmail',
    auth: {
      user: require('./.emailUsername'),
      pass: require('./.emailPassword')
    }
});
```

```
var sendEmail = function(req, callback){
  var urlParts     = req.url.split('/');
  var email        = {
    from:    require('./.emailUsername'),
    to:      decodeURI(urlParts[1]),
    subject: decodeURI(urlParts[2]),
    text:    decodeURI(urlParts[3]),
  };
  transporter.sendMail(email, function(error, info){
    callback(error, email);
  });
};
```

THE SERVER

```
var server = function(req, res){
  var start = Date.now();
  var responseCode = 200;
  var response     = {};

  sendEmail(req, function(error, email){
    response.email = email;

    if(error){
      console.log(error);
      responseCode = 500;
      response.error = error;
    }

    res.writeHead(responseCode, {'Content-Type': 'application/json'});
    res.end(JSON.stringify(response, null, 2));
    var delta = Date.now() - start;
    console.log('Sent an email to ' + email.to + ' in ' + delta + 'ms');
  });

};

http.createServer(server).listen(httpPort, httpHost);
```




DEMO TIME

STRATEGY SUMMARY

- **Why it is better in node:**
  - The client still needs to wait for the message to send, but you won't block any other client's requests
  - Avg response time of ~2 seconds from my couch
- **Why it is still a bad idea:**
  - Slow for the client
  - Spending "web server" resources on sending email
  - Error / Timeout to the client for "partial success"
    - IE: Account created but email not sent
    - Confusing to the user, dangerous for the DB

2) PARALLEL TASKS

IMPROVEMENT IDEAS

- In any other language this would be called “threading”
  - But if it were real threading, the client would still have to wait
  - I guess this might help you catch errors…
  - **note: do not get into a discussion about threads in node…*
- Lets get crazy:
  - **Ignore the Callback**

IGNORE THE CALLBACK

```
var sendEmail = function(req, callback){
  var urlParts     = req.url.split('/');
  var email        = {
    from:    require('./.emailUsername'),
    to:      decodeURI(urlParts[1]),
    subject: decodeURI(urlParts[2]),
    text:    decodeURI(urlParts[3]),
  };
  transporter.sendMail(email, function(error, info){
    if(typeof callback === 'function'){
      callback(error, email);
    }
  });
};
```

```
var server = function(req, res){
  var start = Date.now();
  var responseCode = 200;
  var response     = {};
  sendEmail(req);
  res.writeHead(responseCode, {'Content-Type': 'application/json'});
  res.end(JSON.stringify(response, null, 2));
  console.log('Sent an email');
};


http.createServer(server).listen(httpPort, httpHost);
```


DEMO TIME

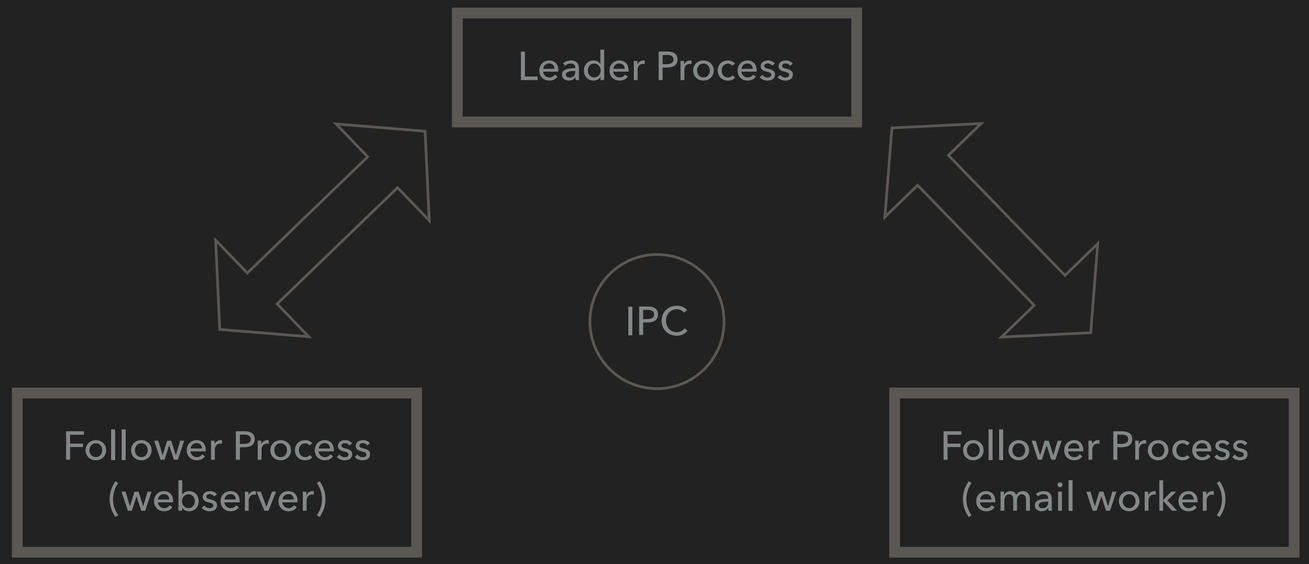
STRATEGY SUMMARY

- **Why it is better in node:**
  - It’s rare you can actually do this in a language… without threading or folks!
  - Crazy-wicked-fast.
- **Why it is still a bad idea:**
  - 0 callbacks, 0 data captured
  - I guess you could log errors?
  - But what would you do with that data?

3) LOCAL MESSAGES

or: “The part of the talk where we grossly over-engineer some stuff”

IMPROVEMENT IDEAS



CLUSTERING IN NODE.JS
A SIMPLE TIMED MANAGER SCRIPT...

```
var cluster = require('cluster');


if(cluster.isMaster){
  doMasterStuff();
}else{
  if(process.env.ROLE === 'server'){ doServerStuff(); }
  if(process.env.ROLE === 'worker'){ doWorkerStuff(); }
}
```

```
var doMasterStuff = function(){
  log('master', 'started master');

  var masterLoop = function(){
    checkOnWebServer();
    checkOnEmailWorker();
  };

  var checkOnWebServer = function(){
  …
  };

  var checkOnEmailWorker = function(){
  …
  };

  setInterval(masterLoop, 1000);
};
```

CLUSTERING IN NODE.JS

```
var doServerStuff = function(){
  var server = function(req, res){
    var urlParts = req.url.split('/');
    var email    = {
      to:      decodeURI(urlParts[1]),
      subject: decodeURI(urlParts[2]),
      text:    decodeURI(urlParts[3]),
    };

    var response = {email: email};
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify(response, null, 2));

    process.send(email);
  };

  http.createServer(server).listen(httpPort, '127.0.0.1');
};
```

Interprocess Communication (IPC) with complex data-types

CLUSTERING IN NODE.JS

```
var checkOnWebServer = function(){
    if(children.server === undefined){
      log('master', 'starting web server');
      children.server = cluster.fork({ROLE: 'server'});
      children.server.name = 'web server';
      children.server.on('online',    function(){ log(children.server, 'ready on port ' + httpPort); });
      children.server.on('exit',      function(){
        log(children.server, 'died :(');
        delete children.server;
      });
      children.server.on('message',   function(message){
        log(children.server, 'got an email to send from the webserver: ' + JSON.stringify(message));
        children.worker.send(message);
      });
    }
  };
```

. . . IT'S ALL JUST MESSAGE PASSING

CLUSTERING IN NODE.JS

```
var checkOnEmailWorker = function(){
    if(children.worker === undefined){
      log('master', 'starting email worker');
      children.worker = cluster.fork({ROLE: 'worker'});
      children.worker.name = 'email worker';
      children.worker.on('online',    function(){ log(children.worker, 'ready!'); });
      children.worker.on('exit',      function(){
        log(children.worker, 'died :(');
        delete children.worker;
      });
      children.worker.on('message',   function(message){
        log(children.worker, JSON.stringify(message));
      });
    }
  };
```

. . . IT'S ALL JUST MESSAGE PASSING

LOCAL MESSAGES

```
var doWorkerStuff = function(){
  process.on('message', function(message){
    emails.push(message);
  });

  var sendEmail = function(to, subject, text, callback){
    …
  };

  var workerLoop = function(){
    if(emails.length === 0){
      setTimeout(workerLoop, 1000);
    }else{
      var e = emails.shift();
      process.send({msg: 'trying to send an email...'});
      sendEmail(e.to, e.subject, e.text, function(error){
        if(error){
          emails.push(e); // try again
          process.send({msg: 'failed sending email, trying again :('});
        }else{
          process.send({msg: 'email sent!'});
        }
        setTimeout(workerLoop, 1000);
      });
    }
  };

  workerLoop();
};
```

Message Queue

Throttling

Retry

Interprocess Communication (IPC) with complex data-types


DEMO TIME

STRATEGY SUMMARY

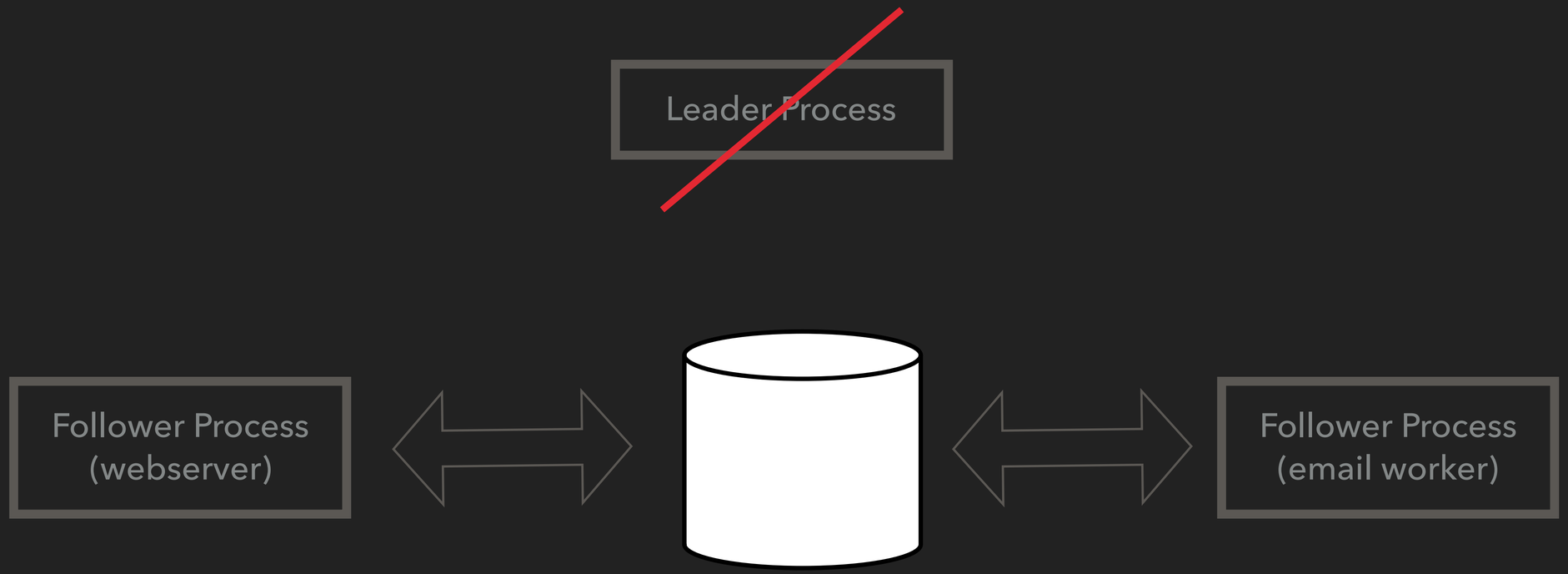
- Notes:
  - the followers never log themselves
    - the leader does it for them
  - Each process has it’s own “main” loop:
    - web server
    - worker
    - leader
  - we can kill the child processes / allow them to crash…

STRATEGY SUMMARY

- **Why it is better in node:**
  - In ~100 lines of JS...
    - Messages aren't lost when server dies
    - Web-server process not bothered by email sending
    - Error handling, Throttling, Queuing and retries!
    - Offline support?
- **Why it is still a bad idea:**
  - Bound to one server

4) REMOTE MESSAGES

IMPROVEMENT IDEAS




node
JS

IPC => REDIS PUB/SUB

```
var doServerStuff = function(){
  var server = function(req, res){
    var urlParts = req.url.split('/');
    var email    = {
      to:      decodeURI(urlParts[1]),
      subject: decodeURI(urlParts[2]),
      text:    decodeURI(urlParts[3]),
    };

    var response = {email: email};
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify(response, null, 2));

    process.send(email);
  };

  http.createServer(server).listen(httpPort, '127.0.0.1');
};
```

```
var doServerStuff = function(){
  var publisher = Redis();

  var server = function(req, res){
    var urlParts = req.url.split('/');
    var email    = {
      to:      decodeURI(urlParts[1]),
      subject: decodeURI(urlParts[2]),
      text:    decodeURI(urlParts[3]),
    };

    publisher.publish(channel, JSON.stringify(email), function(){
      var response = {email: email};
      res.writeHead(200, {'Content-Type': 'application/json'});
      res.end(JSON.stringify(response, null, 2));
    });
  };

  http.createServer(server).listen(httpPort, httpHost);
  console.log('Server running at ' + httpHost + ':' + httpPort);
  console.log('send an email and message to /TO_ADDRESS/SUBJECT/YOUR_MESSAGE');
};
```

REMOTE MESSAGES

```
var doWorkerStuff = function(){
  process.on('message', function(message){
    emails.push(message);
  });

  var sendEmail = function(to, subject, text, callback){
    …
  };

  var workerLoop = function(){
    if(emails.length === 0){
      setTimeout(workerLoop, 1000);
    }else{
      var e = emails.shift();
      process.send({msg: 'trying to send an email...'});
      sendEmail(e.to, e.subject, e.text, function(error){
        if(error){
          emails.push(e); // try again
          process.send({msg: 'failed sending email, trying again :('});
        }else{
          process.send({msg: 'email sent!'});
        }
        setTimeout(workerLoop, 1000);
      });
    }
  };

  workerLoop();
};
```

```
var subscriber = Redis();

  subscriber.subscribe(channel);
  subscriber.on('message', function(channel, message){
    console.log('Message from Redis!');
    emails.push(JSON.parse(message));
  });
```

Still with Throttling and Retry!


DEMO TIME

STRATEGY SUMMARY

- **Why it is better in node:**
  - Redis Drivers are awesome
    - Message Buffering (for connection errors)
    - Thread-pools
    - Good language features (promises and callbacks)
  - Now we can use more than one server!
- **Why it is still a bad idea:**
  - You can only have 1 type of server

5) REMOTE QUEUE

IMPROVEMENT IDEAS

- Observability
  - How long is the queue?
  - How long does an item wait in the queue?
  - Operational Monitoring
- Redundancy
  - Backups
  - Clustering


node
JS

DATA STRUCTURES NEEDED FOR AN MVP QUEUE

- Array
  - push, pop, length

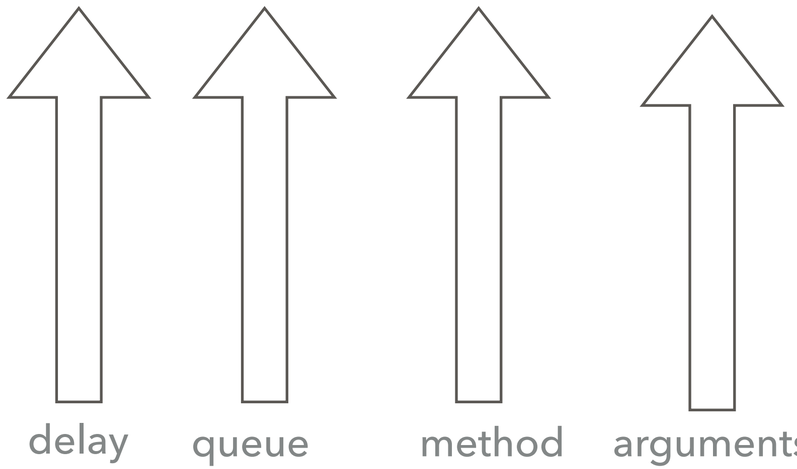
DATA STRUCTURES NEEDED FOR A GOOD QUEUE

- Array
  - push, pop, length
- Hash (key types: string, integer, hash)
  - Set, Get, Exists
- Sorted Set
  - Exists, Add, Remove

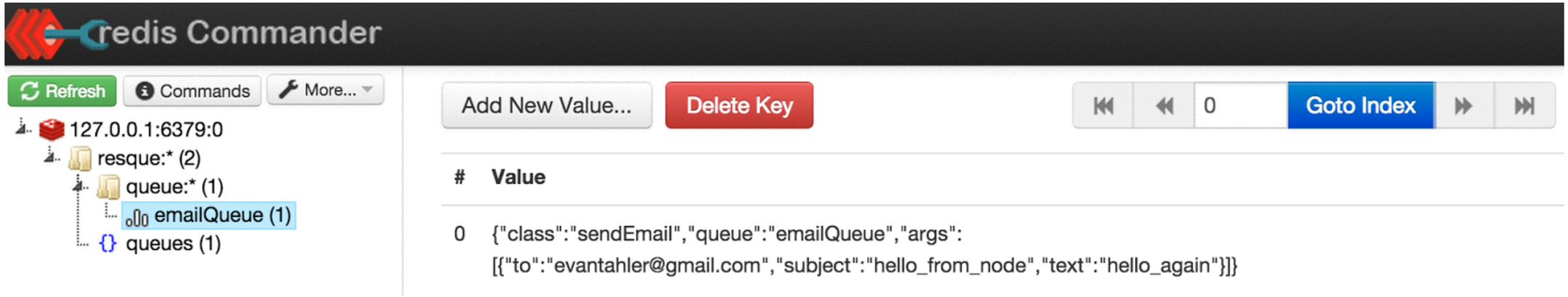
REDIS HAS THEM ALL!

RESQUE: DATA STRUCTURE FOR QUEUES IN REDIS

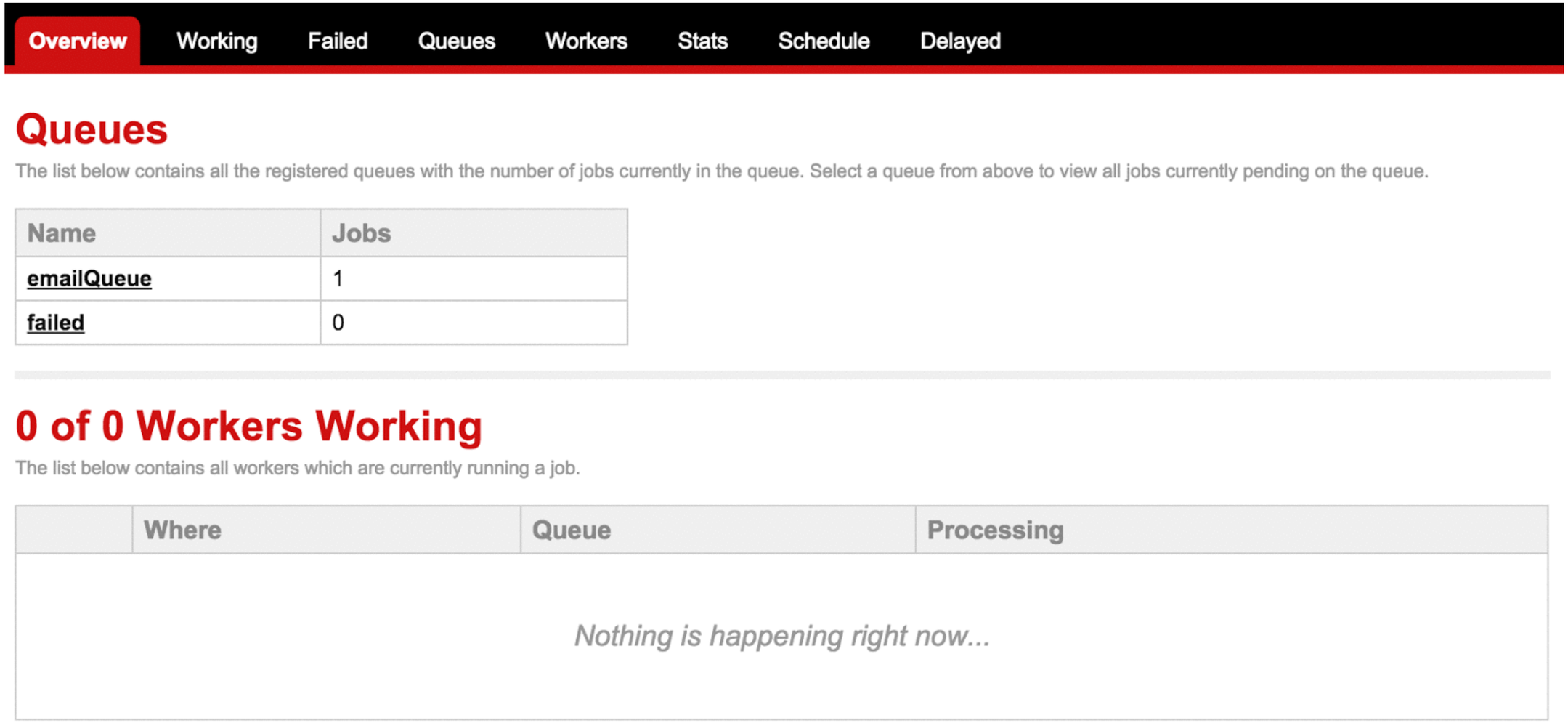
```
var NR = require('node-resque');
var queue = new NR.queue({connection: connectionDetails}, jobs);

queue.on('error', function(error){ console.log(error); });

queue.connect(function(){
  queue.enqueue('math', "add", [1,2]);
  queue.enqueueIn(3000, 'math', "subtract", [2,1]);
});
```



RESQUE: DATA STRUCTURE FOR QUEUES IN REDIS



RESQUE: DATA STRUCTURE FOR QUEUES IN REDIS



USING NODE-RESQUE

```
var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: require('./.emailUsername'),
    pass: require('./.emailPassword')
  }
});

var jobs = {
  sendEmail: function(data, callback){
    var email = {
      from:    require('./.emailUsername'),
      to:      data.to,
      subject: data.subject,
      text:    data.text,
    };

    transporter.sendMail(email, function(error, info){
      callback(error, {email: email, info: info});
    });
  }
};
```

SENDING EMAILS IS A “JOB” NOW

USING NODE-RESQUE

```
var server = function(req, res){
  var urlParts = req.url.split('/');
  var email    = {
    to:      decodeURI(urlParts[1]),
    subject: decodeURI(urlParts[2]),
    text:    decodeURI(urlParts[3]),
  };

  queue.enqueue('emailQueue', "sendEmail", email, function(error){
    if(error){ console.log(error) }
    var response = {email: email};
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify(response, null, 2));
  });
};



var queue = new NR.queue({connection: connectionDetails}, jobs);
queue.connect(function(){
  http.createServer(server).listen(httpPort, httpHost);
  console.log('Server running at ' + httpHost + ':' + httpPort);
  console.log('send an email and message to /TO_ADDRESS/SUBJECT/YOUR_MESSAGE');
});
```

USING NODE-RESQUE

```
var worker = new NR.worker({connection: connectionDetails, queues: ['emailQueue']}, jobs);
worker.connect(function(){
  worker.workerCleanup();
  worker.start();
});

worker.on('start',           function(){ console.log("worker started"); });
worker.on('end',             function(){ console.log("worker ended"); });
worker.on('cleaning_worker', function(worker, pid){ console.log("cleaning old worker " + worker); });
worker.on('poll',            function(queue){ console.log("worker polling " + queue); });
worker.on('job',             function(queue, job){ console.log("working job " + queue + " " + JSON.stringify(job)); });
worker.on('reEnqueue',       function(queue, job, plugin){ console.log("reEnqueue job (" + plugin + ") " + queue + " " + JSON.stringify(job)); });
worker.on('success',         function(queue, job, result){ console.log("job success " + queue + " " + JSON.stringify(job) + " >> " + result); });
worker.on('failure',         function(queue, job, failure){ console.log("job failure " + queue + " " + JSON.stringify(job) + " >> " + failure); });
worker.on('error',           function(queue, job, error){ console.log("error " + queue + " " + JSON.stringify(job) + " >> " + error); });
worker.on('pause',           function(){ console.log("worker paused"); });
```


DEMO TIME

BUT WHAT IS SO SPECIAL ABOUT NODE.JS HERE?

IMPROVEMENT IDEAS

- The node.js event loops is great for processing all non-blocking events, not just web servers.
- Most Background jobs are non-blocking events
  - Update the DB, Talk to this external service, etc
- So node can handle many of these at once per process!
- Redis is fast enough to handle many “requests” from the same process in this manner
  - We can use the same connection or thread-pool

USING NODE-RESQUE AND MAXIMIZING THE EVENT LOOP

```
var multiWorker = new NR.multiWorker({
  connection: connectionDetails,
  queues: ['slowQueue'],
  minTaskProcessors: 1,
  maxTaskProcessors: 20,
}, jobs);
```

non-blocking

```
var jobs = {
  "slowSleepJob": {
    plugins: [],
    pluginOptions: {},
    perform: function(callback){
      var start = new Date().getTime();
      setTimeout(function(){
        callback(null, (new Date().getTime() - start) );
      }, 1000);
    },
  },

  "slowCPUJob": {
    plugins: [],
    pluginOptions: {},
    perform: function(callback){
      var start = new Date().getTime();
      blockingSleep(1000);
      callback(null, (new Date().getTime() - start) );
    },
  },
};
```

blocking

```
var blockingSleep = function(naptime){
  var sleeping = true;
  var now = new Date();
  var alarm;
  var startingMSeconds = now.getTime();
  while(sleeping){
    alarm = new Date();
    var alarmMSeconds = alarm.getTime();
    if(alarmMSeconds - startingMSeconds > naptime){ sleeping = false; }
  }
};
```

HOW CAN YOU TELL IF THE EVENT LOOP IS BLOCKED?

```
// inspired by https://github.com/tj/node-blocked

module.exports = function(limit, interval, fn) {
  var start = process.hrtime();

  setInterval(function(){
    var delta = process.hrtime(start);
    var nanosec = delta[0] * 1e9 + delta[1];
    var ms = nanosec / 1e6;
    var n = ms - interval;
    if (n > limit){
      fn(true, Math.round(n));
    }else{
      fn(false, Math.round(n));
    }
    start = process.hrtime();
  }, interval).unref();
};
```

. . . SEE HOW LONG IT TAKES FOR THE NEXT “LOOP”


DEMO TIME

REDIS

- Redis has unique properties that make it **perfect** for this type of workload
  - FAST
  - Single-threaded so you can have real array operations (pop specifically)
  - Data-structure creation on the fly (new queues)
  - Dependent on only RAM and network

STRATEGY SUMMARY

- **Why it is better in node:**
  - In addition to persistent storage and multiple server/process support, you get get CPU scaling and Throttling very simply!
  - Integrates well with the resque/sidekiq ecosystem
- **This is now a good idea!**

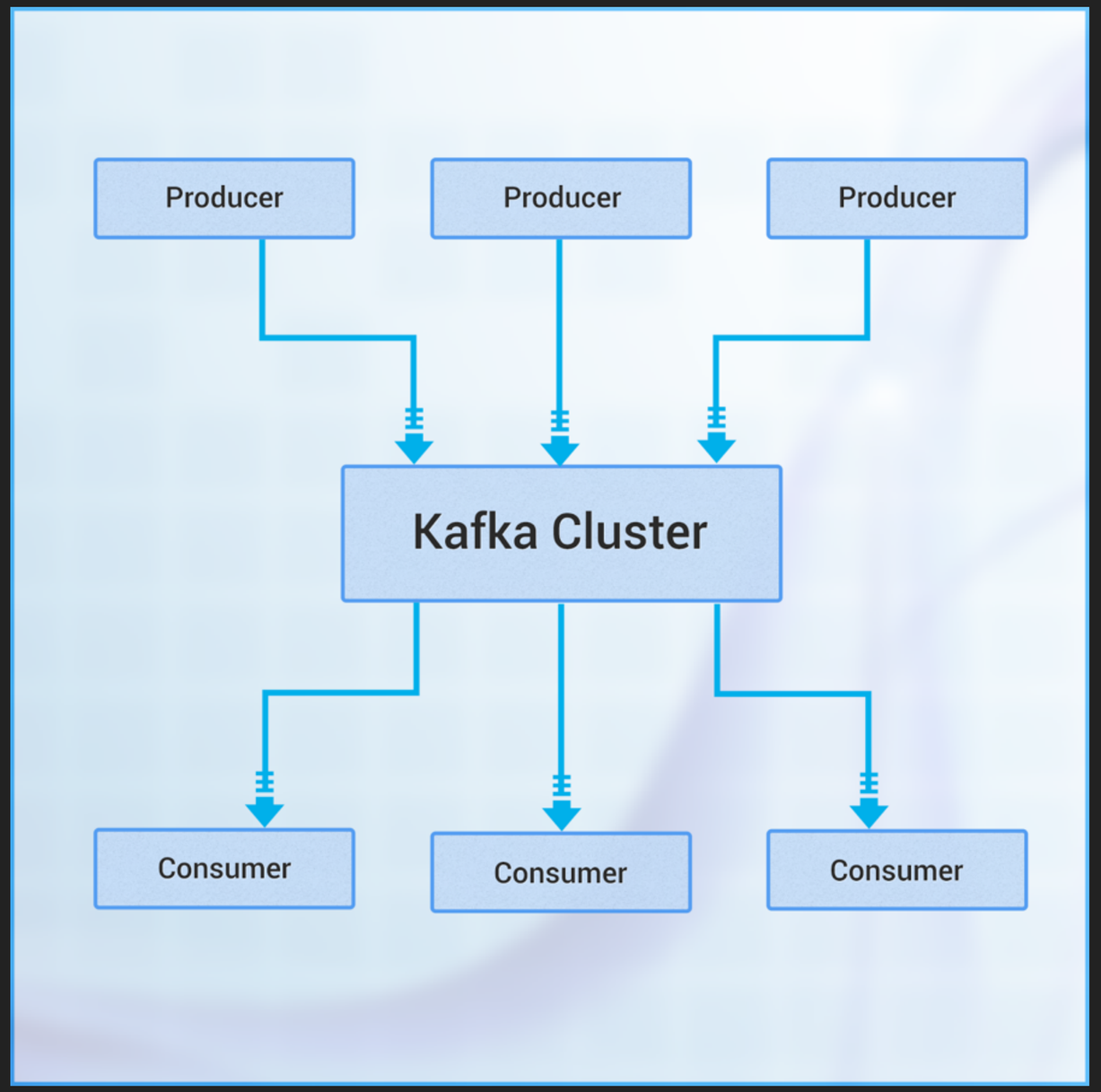
6) IMMUTABLE EVENT BUS

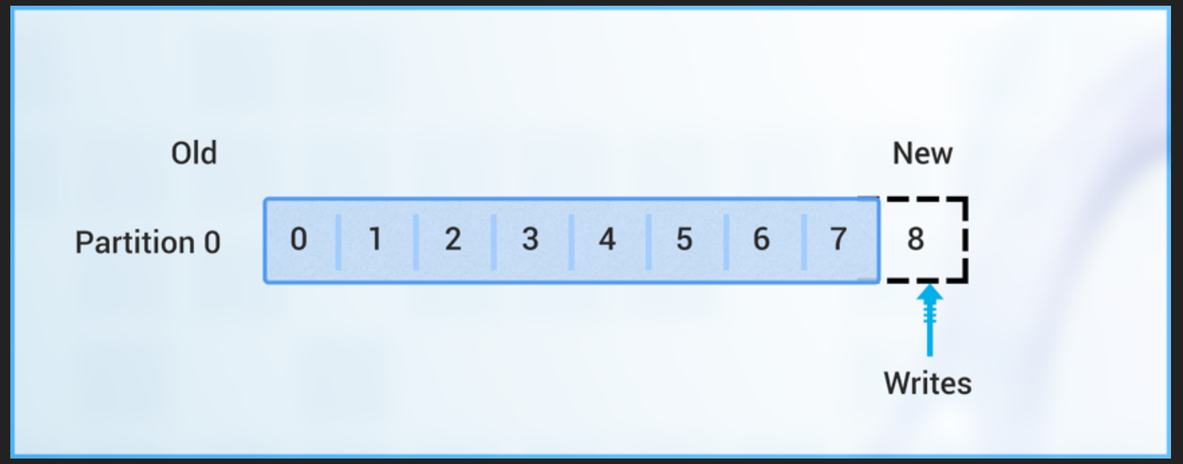
“The Future”… “Maybe”

IMPROVEMENT IDEAS

- What was wrong with the resque pattern?
  - Jobs are consumed and deleted… no historical introspection
  - In redis, storing more and more events to a single key gains nothing from redis-cluster
  - If you wanted to do more than one thing on user#create, you would need to fire many jobs
  - What if we just fired a “user_created” event and let the workers choose what to do?

IMMUTABLE EVENT BUS





- Events are written to a list and never removed
- Consumers know where their last read was and can continue

IMPROVEMENT IDEAS

- What to we need that redis cannot do natively
  - A “blocking” get and incr
  - Tools to seek for “the next key” for listing partitions

. . . GOOD THING WE HAVE LUA!

LUA LIKE A BOSS...

```
local name = ARGV[1]
local partition = "eventr:partitions:" .. ARGV[2]
local coutnerKey = "eventr:counters:" .. ARGV[2]

-- If we have a key already for this name, look it up
local counter = 0
if redis.call("HEXISTS", coutnerKey, name) == 1 then
  counter = redis.call("HGET", coutnerKey, name)
  counter = tonumber(counter)
end

-- if the partition exists, get the data from that key
if redis.call("EXISTS", partition) == 0 then
  return nil
else
  local event = redis.call("LRANGE", partition, counter, counter)
  if (event and  event[1] ~= nil) then
    redis.call("HSET", coutnerKey, name, (counter + 1))
    return event[1]
  else
    return nil
  end
end
```

SENDING EMAILS IS NOW A HANDLER

```
var handlers = [
  {
    name: 'email handler',
    perform: function(event, callback){
      if(event.eventName === 'emailEvent'){
        var email = {
          from:    require('./.emailUsername'),
          to:      event.email.to,
          subject: event.email.subject,
          text:    event.email.text,
        };

        transporter.sendMail(email, function(error, info){
          callback(error, {email: email, info: info});
        });
      }else{
        return callback();
      }
    }
  }
];

var consumer = new eventr('myApp', connectionDetails, handlers);

consumer.work();
```

THE WEB SERVER PUBLISHES EVENTS

```
var server = function(req, res){
  var urlParts = req.url.split('/');
  var email    = {
    to:      decodeURI(urlParts[1]),
    subject: decodeURI(urlParts[2]),
    text:    decodeURI(urlParts[3]),
  };

  producer.write({
    eventName: 'emailEvent',
    email: email
  }, function(error){
    if(error){ console.log(error) }
    var response = {email: email};
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify(response, null, 2));
  });
};

var producer = new eventr('webApp', connectionDetails);
http.createServer(server).listen(httpPort, httpHost);
```

CONSUMER STEP 1: WHAT PARTITION ARE WE WORKING ON?

```
jobs.push(function(next){
    self.redis.hget(self.prefix + 'client_partitions', self.name, function(error, p){
      if(error){ return next(error); }
      if(!p){
        self.firstPartition(function(error, p){
          if(error){ return next(error); }
          if(p){ partition = p; }
          self.redis.hset(self.prefix + 'client_partitions', self.name, partition, next);
        })
      }else{
        partition = p;
        return next();
      }
    });
  });
```

CONSUMER STEP 2: ARE THERE ANY EVENTS?

```
jobs.push(function(next){
    if(!partition){ return next(); }
    self.emit('poll', partition);
    self.redis.getAndIncr(self.name, partition, function(error, e){
      if(error){ return next(error); }
      if(e){ event = JSON.parse(e); }
      return next();
    });
  });
```

The `ioredis` package is awesome!

```
  var lua = fs.readFileSync(__dirname + '/6-lua.lua').toString();

  this.redis.defineCommand('getAndIncr', {
    numberOfKeys: 0,
    lua: lua
  });
```

CONSUMER STEP 3: TRY THE NEXT PARTITION

```
jobs.push(function(next){
    if(!partition){ return next(); }
    if(event){ return next(); }
    self.nextPartition(partition, function(error, nextPartition){
      if(nextPartition){
        self.redis.hset(self.prefix + 'client_partitions', self.name, nextPartition, next);
      }else{
        next();
      }
    });
  });
```

```
eventr.prototype.nextPartition = function(partition, callback){
  var self = this;
  self.redis.zrank(self.prefix + 'partitions', partition, function(error, position){
    if(error){ return callback(error); }
    self.redis.zrange(self.prefix + 'partitions', (position + 1), (position + 1), function(error, partitions){
      if(error){ return callback(error); }
      return callback(null, partitions[0]);
    });
  });
}
```

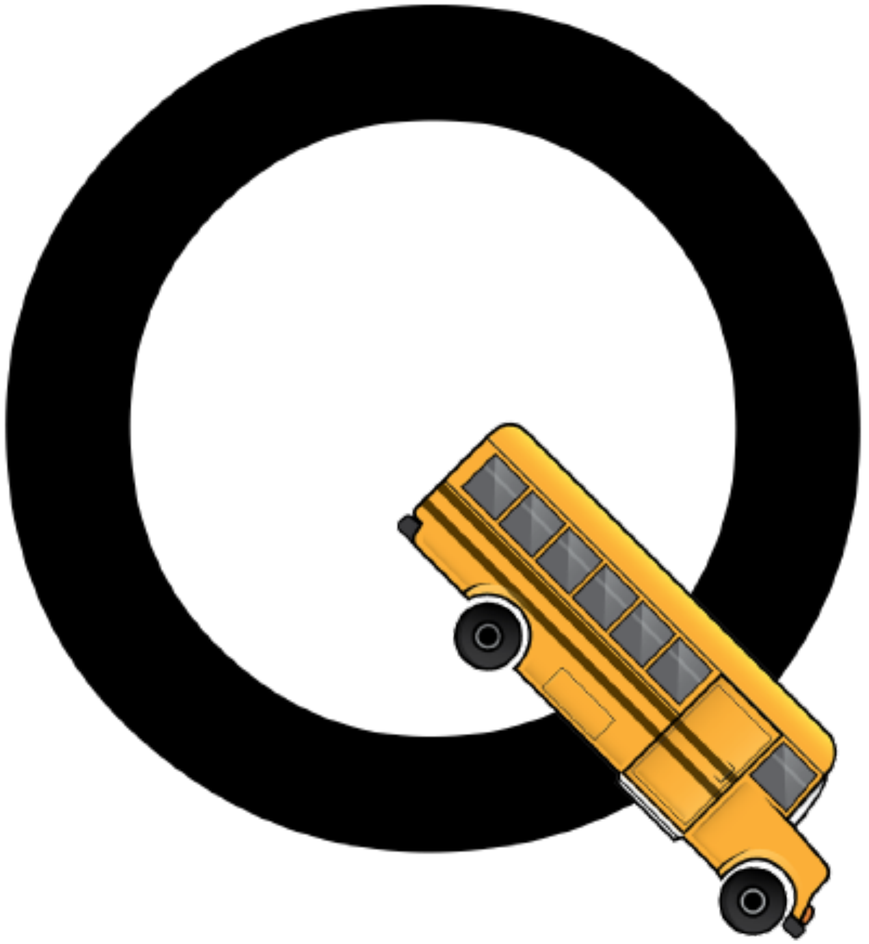

DEMO TIME

STRATEGY SUMMARY

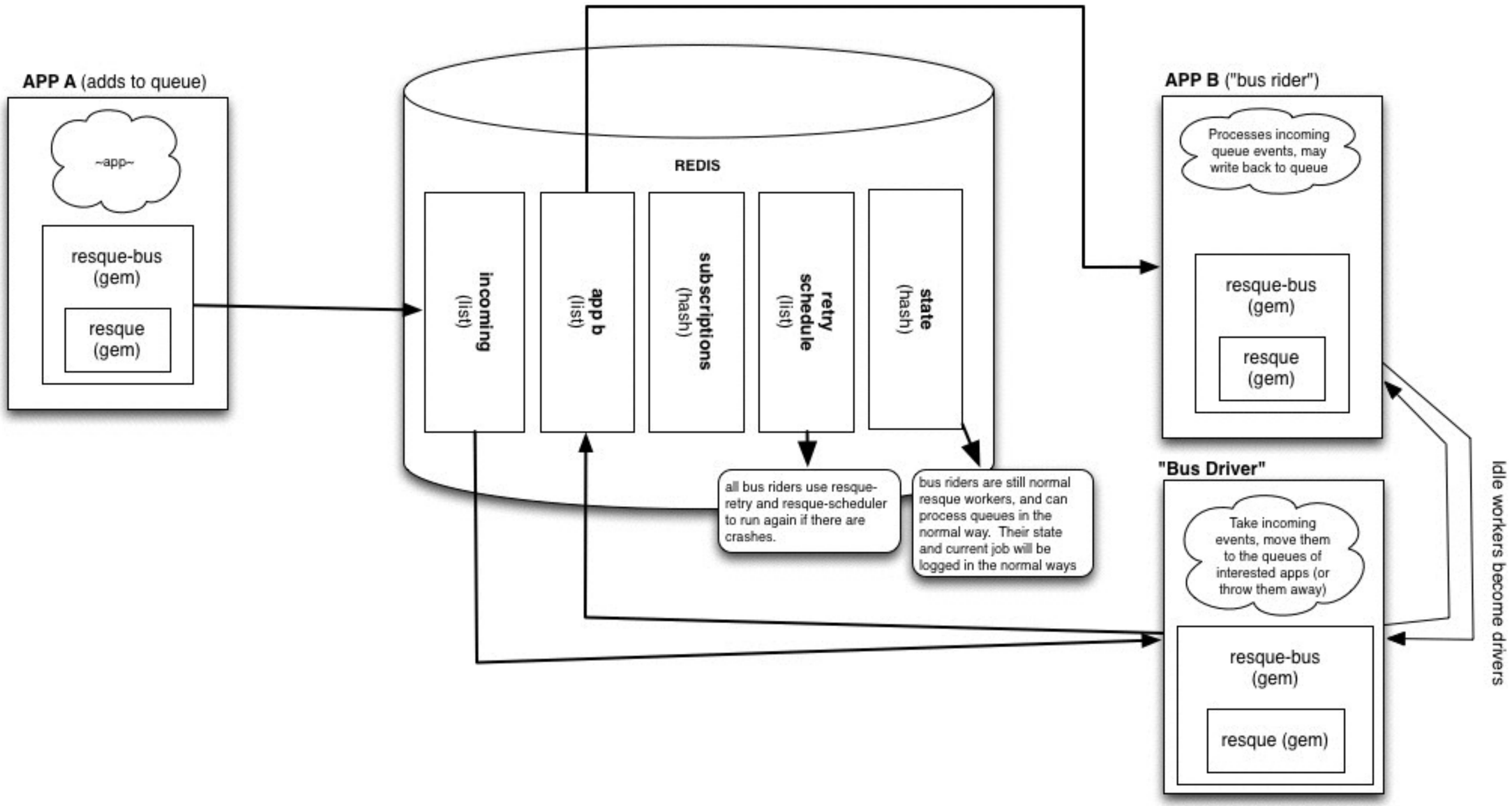
- **Why it is better in node:**
  - Just like with Resque, we can poll/stream many events at once (non-blocking).
  - We can now make use of Redis Cluster's key distribution
- **Is this a bad idea?**
  - Maybe.
  - We would need a lot of LUA. We are actively slowing down Redis to preform more operations in a single block.
  - Do we really need to store our history in Ram? Wouldn't disk be better?
  - We'll need to delete the old keys after a while

CAN WE WE MEET IN THE MIDDLE OF “IMMUTABLE EVENT BUS” + “RESQUE”?

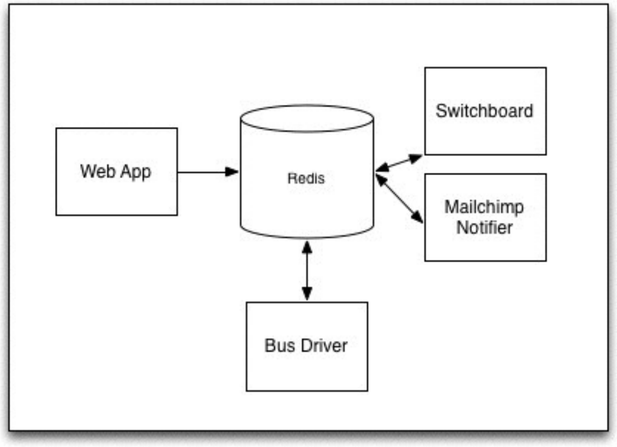
QUEUEBUS

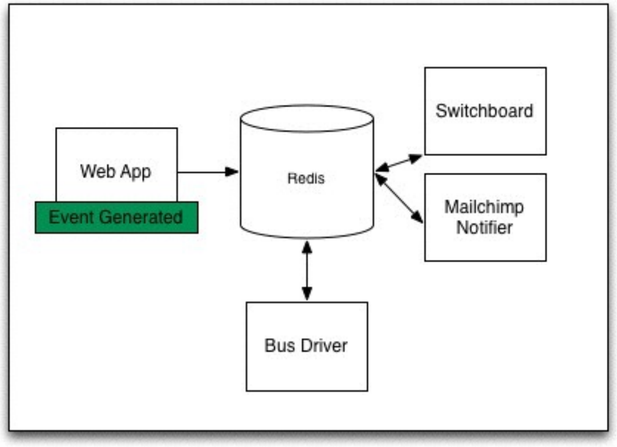


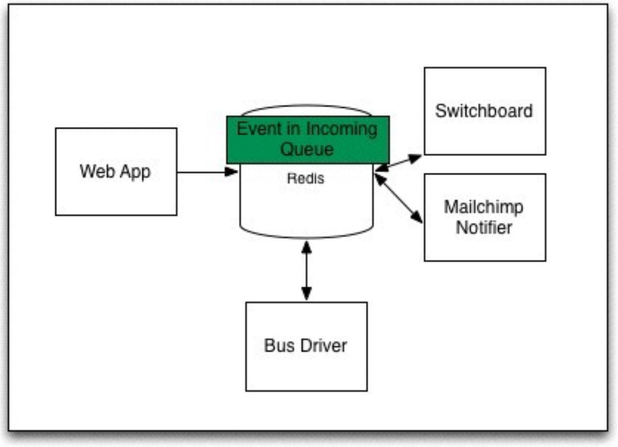
QUEUEBUS

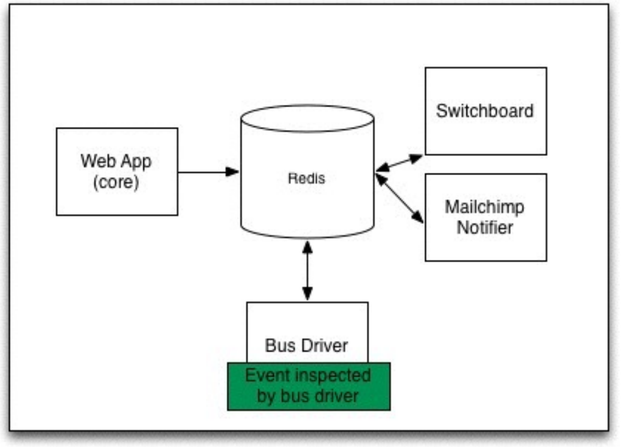

APP A (adds to queue)
~app~
resque-bus (gem)
resque (gem)
REDIS
incoming (list)
app b (list)
subscriptions (hash)
retry schedule (list)
state (hash)
all bus riders use resque-retry and resque-scheduler to run again if there are crashes.
bus riders are still normal resque workers, and can process queues in the normal way. Their state and current job will be logged in the normal ways
APP B ("bus rider")
Processes incoming queue events, may write back to queue
resque-bus (gem)
resque (gem)
"Bus Driver"
Take incoming events, move them to the queues of interested apps (or throw them away)
resque-bus (gem)
resque (gem)
Idle workers become drivers

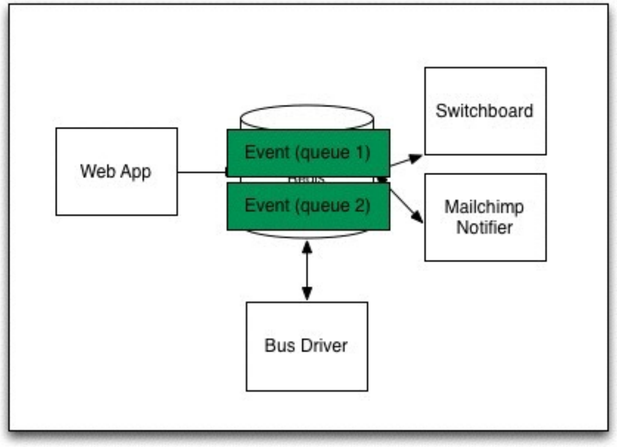
QUEUEBUS

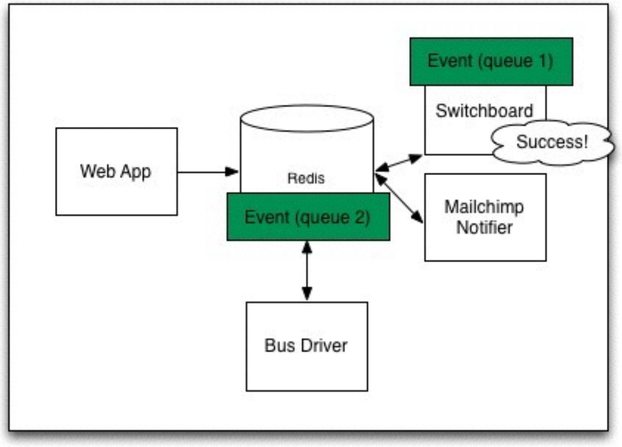

Web App
Redis
Switchboard
Mailchimp Notifier
Bus Driver

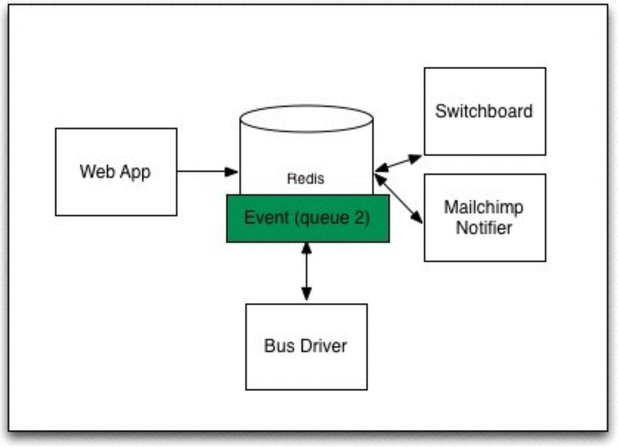

Web App
Event Generated
Redis
Switchboard
Mailchimp Notifier
Bus Driver

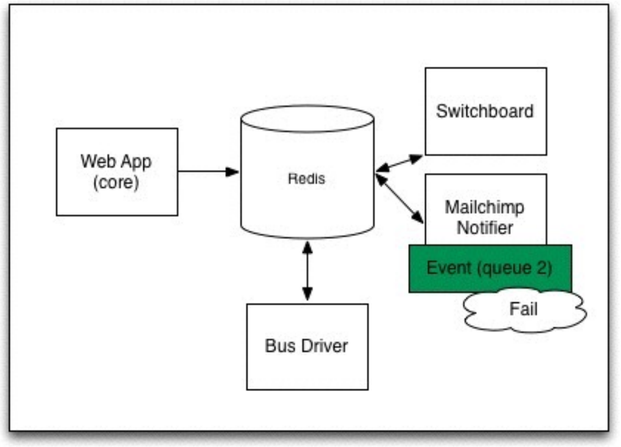

Web App
Event in Incoming Queue
Redis
Switchboard
Mailchimp Notifier
Bus Driver


Web App (core)
Redis
Switchboard
Mailchimp Notifier
Bus Driver
Event inspected by bus driver


Web App
Event (queue 1)
Event (queue 2)
Switchboard
Mailchimp Notifier
Bus Driver


Web App
Redis
Event (queue 1)
Event (queue 2)
Switchboard
Success!
Mailchimp Notifier
Bus Driver


Web App
Redis
Event (queue 2)
Switchboard
Mailchimp Notifier
Bus Driver


Web App (core)
Redis
Switchboard
Mailchimp Notifier
Event (queue 2)
Fail
Bus Driver

STRATEGY SUMMARY

- **Why it is better in node:**
  - Just like with Resque, we can poll/stream many events at once (non-blocking)
- **What did we gain over Resque?**
  - Syndication of events to multiple consumers
- **What didn’t we get from Kafka?**
  - re-playable event log

POSSIBLE TASK STRATEGIES:

FOREGROUND (IN-LINE)

PARALLEL (THREAD-ISH)

LOCAL MESSAGES (FORK-ISH)

REMOTE MESSAGES (*MQ-ISH)

REMOTE QUEUE (REDIS + RESQUE)

IMMUTABLE EVENT BUS (KAFKA-ISH)

THANKS!

IOREDIS: GITHUB.COM/LUIN/IOREDIS

NODE-MAILER: GITHUB.COM/NODEMAILER/NODEMAILER

ASYNC: GITHUB.COM/CAOLAN/ASYNC

BACKGROUND TASKS IN NODE.JS

NODE-RESQUE:

https://github.com/taskrabbit/node-resque

QUEUE-BUS:

https://github.com/queue-bus

SUPPORTING PROJECT + SLIDES:

https://github.com/evantahler/background_jobs_node

$20 OFF TASKRABBIT

http://bit.ly/evan-tr

@EVANTAHLER