BUILDING NODE APPS THAT SCALE

(OR -- NODE IS WEB SCALE!)

Brandon Mayes

May 12, 2016

SPANNING?

Spanning === SaaS data protection === Granular, application-aware backup and recovery for cloud applications

APPS WE SUPPORT

- Google Apps
- Salesforce
- Office 365

Mostly due to user error, but you actually do need to back these things up.

THREE MAIN ACTIONS

- Backup
- Restore
- Export

Product obviously does more than this but these are the big ones.

SOME GENERAL CHALLENGES

- Scalable to handle as many concurrent actions as necessary
- Fault-tolerant (retry logic, resumability, etc.)
- Actions may take weeks to complete
- Order of operations matters (but Node is async!?)
- Point-in-time with unlimited storage

SOME NOTES ON BACKUPS

- 250GB+ of content stored every week in S3
- 150M+ new rows every week to our largest DB table (and growing)
- Guaranteed to run every day
- Must run in < 24 hours
  - We would prefer it to be MUCH less
  - Exception for initial backup

WHY THIS TALK?

Creating a scalable, fault-tolerant, enterprise-ready application requires significant time and effort.

Can't just turn it on and have it scale right up. Lambda might work for some stuff but can't imagine an entire application. How would you handle state, errors, retry logic, etc. in an entire lambda app?

WHAT WE'LL COVER

- Separate your concerns
- Promise all the things!
- Test all the things!
- Monitor all the things!
- Areas we have struggled with/need improvement

1. Stuff that works for us (maybe !everyone)
2. Watching from different angles:
- How you think about the things vs. do the things vs. watch the things

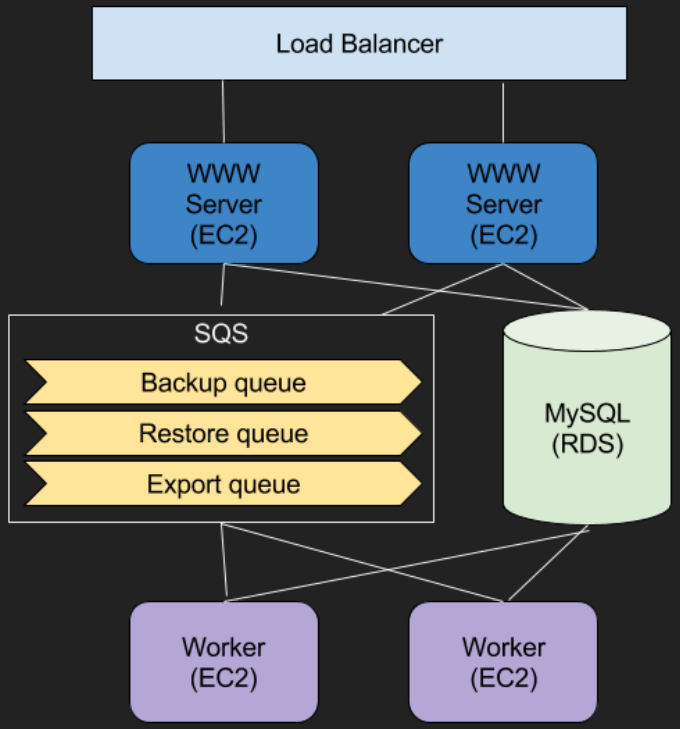
SEPARATE YOUR CONCERNS

- We run a somewhat SOA-like implementation.
- Easiest to start with an overview of our architecture...

Not SOA purists so there isn't a dedicated API for *everything*

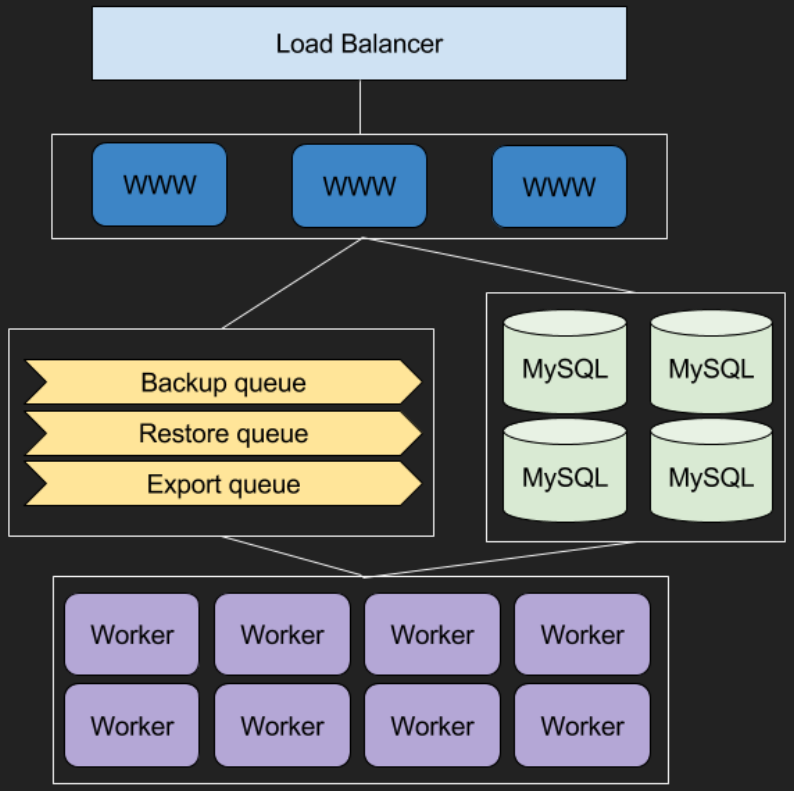
This is where I Rickrolled everyone. If you're reading these slides and somehow aren't familiar with this term, see https://en.wikipedia.org/wiki/Rickrolling for more information.

EXAMPLE ARCHITECTURE



Simplified view, and not cloudy (not scalable)

SCALABLE ARCHITECTURE



More representative of our actual environment. Different tiers of machines (SOA)
- Scale out in a horizontal fashion independently of other tiers
- DB tier is our "least cloudy" requiring manual work to add more space

WEB TIER

- Scales up/down with auto-scale rules
- Not very interesting

CAT GIF!

Much more interesting...

WORKER TIER

- EC2 instances running X Node processes that poll SQS
  - Some always online, others scale up and down
  - On-demand vs. scheduled backup queues
  - Can monitor many queues in priority order
- Most elastic tier
- Also scale up/down with auto-scale rules
- Also not very interesting

Most elastic tier but also roughly predictable. A few dedicated worker processes are always running.
Queue up all automated daily backups then auto-scale rules kick in and bring us to max size. We step down to zero as queue burns down.

A QUICK GLOSSARY

- Batch: A unique job within our system (datetime)
- Salesforce (SF) ID: 18-char UUID assigned to every record
- Object (SObject): Type of a record in Salesforce hierarchy
- Org: Salesforce organization (basically, a single customer)

DB SCALABILITY

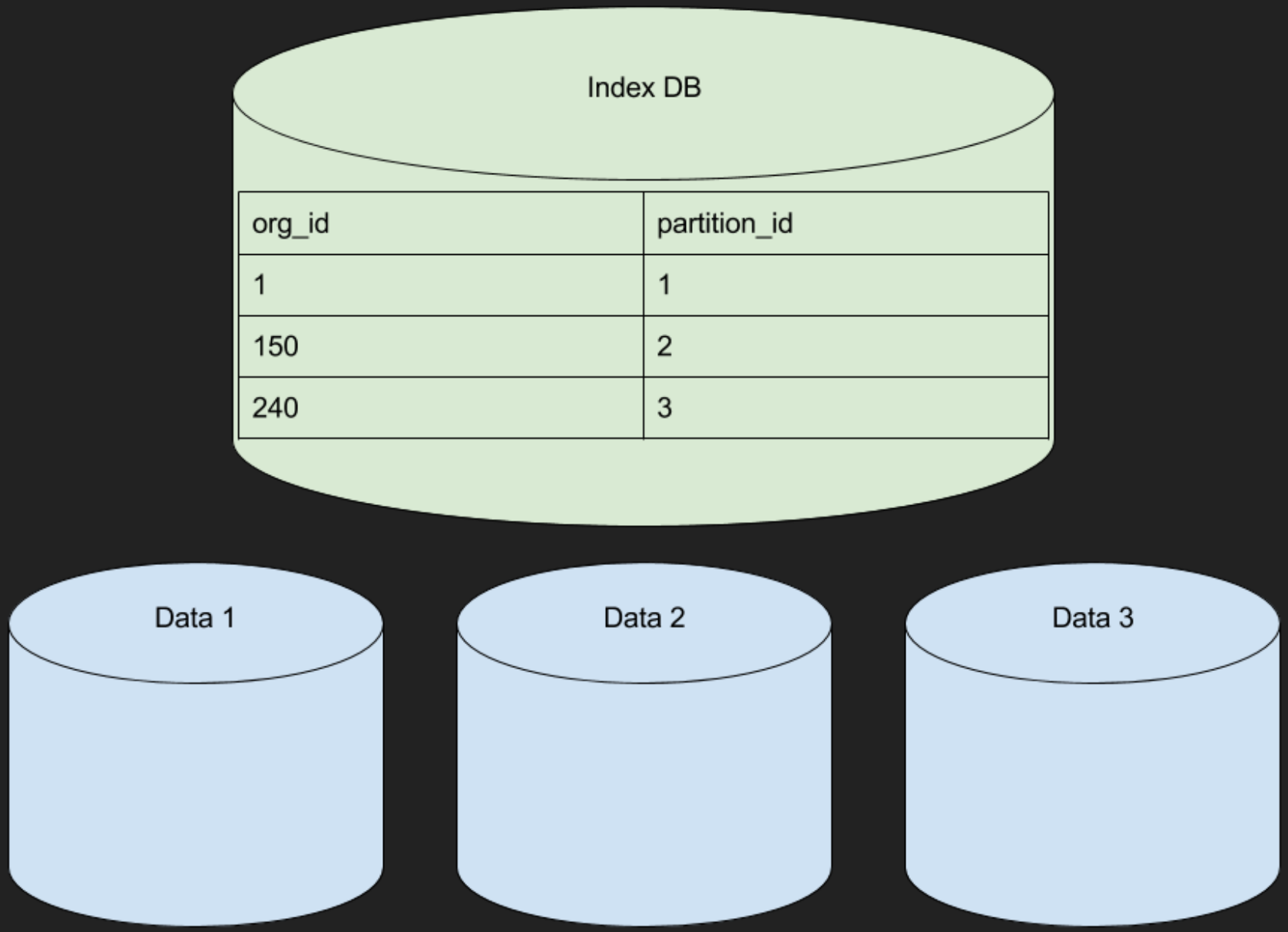
Why not just use NoSQL?

- Hot spots if primary key (hash) isn't chosen correctly
  - Initial backup batches can include millions of records for one object type
  - Some customers change every object every day
- Often doesn't provide ability to index data easily
- Lacks rich query semantics:
  - find updated and/or deleted records for org 12 in Tuesday's backup batch
  - find active Account records for org 73 that existed in backup batch on 12/22/2014

O365 team migrating from Cassandra to Postgres as NoSQL doesn't fit our use cases (query patterns) very well.

MYSQL WITH MANUAL SHARDING

- Single "index" database (for now)
- Index points to data partition where org's data resides



All data partitions are separate, reserved RDS instances. They have identical schema across all of them however.

SORT OF WORKED...

- One table in particular became ridiculously large
- Had to write data migrator to copy data for large orgs to "smaller" partitions
- Eventually one org will max out an RDS database on their own (unlimited retention, remember?)

ANOTHER LAYER OF SHARDING

- Daily process to calculate partition "weights" based on partition's bytes used vs. average across partition
- Backups consult weight table and pick a partition at start
- Lots of refactoring (abstraction) around DAO layer to query across partitions and aggregate results
- We'll probably have to scatter other tables like this soon

Cool pattern that we had, so take it to the next level.
- Because of point-in-time semantics we can cap the growth of any particular partition.
- Uniform/predictable way to horizontally scale.

But enough about databases...this is a Node meetup, right?

PROMISE ALL THE THINGS!

- Clean, readable code that feels almost synchronous in nature
- Simplified/uniform error handling (BONUS: retry logic greatly simplified)
- Concurrent execution simplified and managed for you
- Promisify another library
- Mocking for unit tests is still trivial

CALLBACK HELL

```
var p_client = new Db('integration_tests_20', new Server("127.0.0.1",
p_client.open(function(err, p_client) {
    p_client.dropDatabase(function(err, done) {
        p_client.createCollection('test_custom_key', function(err, co
            collection.insert({'a':1}, function(err, docs) {
                collection.find({'_id':new ObjectID("aaaaaaaaaaaa")},
                    cursor.toArray(function(err, items) {
                        test.assertEquals(1, items.length);

                        // Let's close the db
                        p_client.close();
                    });
                });
            });
        });
    });
```

From node-mongodb-native. I pulled this example from http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/

ISN'T THIS SO MUCH BETTER?

```
var p_client = new Db('integration_tests_20', new Server("127.0.0.1",
return p_client.open()
  .then(function(p_client) {
    return p_client.dropDatabase();
  })
  .then(function() {
    return p_client.createCollection('test_custom_key');
  })
  .then(function(collection) {
    return collection.insert({'a':1});
  })
  // remaining lines here
  .catch(function(err) {
    // optional if you want to handle stuff here
  });
```

Much more readable and error handling is great (most people don't argue against this). But concurrency is really where bluebird shines.

HOW ABOUT CONCURRENT I/O?

```
var reapS3Promise = ReaperService.reapS3Content(org.id),
  reapDbPromise = ReaperService.reapDbContent(org.id);

return Promise.join(reapS3Promise, reapDbPromise)
  .spread(function(s3Result, dbResult) {
    logger.info('End reaping for orgId=[%d]', org.id);

    // mark as reaped if both S3 and DB were reaped successfully
    if (updateFlag && s3Result === true && dbResult === true) {
      return OrgService.setOrgAsReaped(org.sfId);
    }
  });
```

Managing and orchestrating concurrency of two separate tasks that do not depend on one another. Only execute code when they are both done.

OR AN UNKNOWN NUMBER OF PROMISES

Example: discover parent-child hierarchy for every object

```
// somehow we get an array of objectTypes
// looks like ['Account', 'Attachment', 'CustomType1', ...]

return Promise.map(objectTypes, function(objectType) {
  return SalesforceService.describeType(org.id, sobjectType)
    .then(function(description) {
      // inspect description.childRelationships to build hierarchy
    });
}, {
  concurrency: 5
});
```

Concurrently perform any number of tasks for a variable length array. Tell me when they are all done.
- May not even care about returned values. Just run a bunch of async things in parallel.

TEST ALL THE THINGS!

- Use linting tool for gotchas and enforcing standards
- LOTS of unit tests! We use:
  - mocha
  - chai
  - sinon (and sinon-as-promised)
- Integration tests too!

- Lint tool because JavaScript is dynamically typed and also to enforce common coding standards.
- I prefer unit tests because speed!
- App is bigger than anything we can fit in our heads.

UNIT TEST EXAMPLE (WITH MOCKS)

```
describe('fake unit tests for foo() function', function() {
  // setup some data and some stubs
  var someFunctionStub, writeDbRecordStub;

  beforeEach(function(testDone) {
    someFunctionStub = sinon.stub(SomeModule, 'someFunction');
    writeDbRecordStub = sinon.stub(SomeModule, 'writeDbRecord');
    testDone();
  });

  afterEach(function(testDone) {
    someFunctionStub.restore();
    writeDbRecordStub.restore();
    testDone();
  });
```

A POSITIVE TEST

```
it('should write to DB with arg >= 5', function(testDone) {
  someFunctionStub.returns(Promise.resolve());
  writeDbRecordStub.returns(Promise.resolve(someValue));

  SomeModule.foo(42)
    .then(function(result) {
      // the result should be 28 for whatever reason
      result.should.equal(28);

      // this stub passed array [1, 2, 3] (used next slide)
      someFunctionStub.callCount.should.equal(1);
      writeDbRecordStub.callCount.should.equal(1);
    })
    .done(testDone, testDone);
});
```

You might also call this happy path testing.

VERIFYING CALL ARGS

```
it('should write to DB with arg >= 5', function(testDone) {
  someFunctionStub.returns(Promise.resolve());
  writeDbRecordStub.returns(Promise.resolve(someValue));

  SomeModule.foo(42)
    .then(function(result) {
      var callArgs;
      result.should.equal(28);
      someFunctionStub.callCount.should.equal(1);
      callArgs = someFunctionStub.getCall(0).args;
      callArgs.length.should.equal(1);
      callArgs[0].should.be.an('array').and.have.members([1, 2, 3]);
      // repeat similar pattern for the writeDbRecordStub
    })
    .done(testDone, testDone);
});
```

A NEGATIVE TEST

```
it('should reject if arg is less than 5', function(testDone) {
  someFunctionStub.rejects(new Error('arg is less than 5'));

  SomeModule.foo(4)
    .then(function() {
      assert(false, 'should not have made it here with arg < 5');
    })
    .catch(function(err) {
      should.exist(err);
      err.message.should.equal('arg is less than 5');

      someFunctionStub.callCount.should.equal(0);
      writeDbRecordStub.callCount.should.equal(0);
    })
    .done(testDone, testDone);
});
```

MONITOR ALL THE THINGS!

- DevOps monitors their metrics and environment
- Product team monitors info to help troubleshoot

DEVOPS MONITORING:

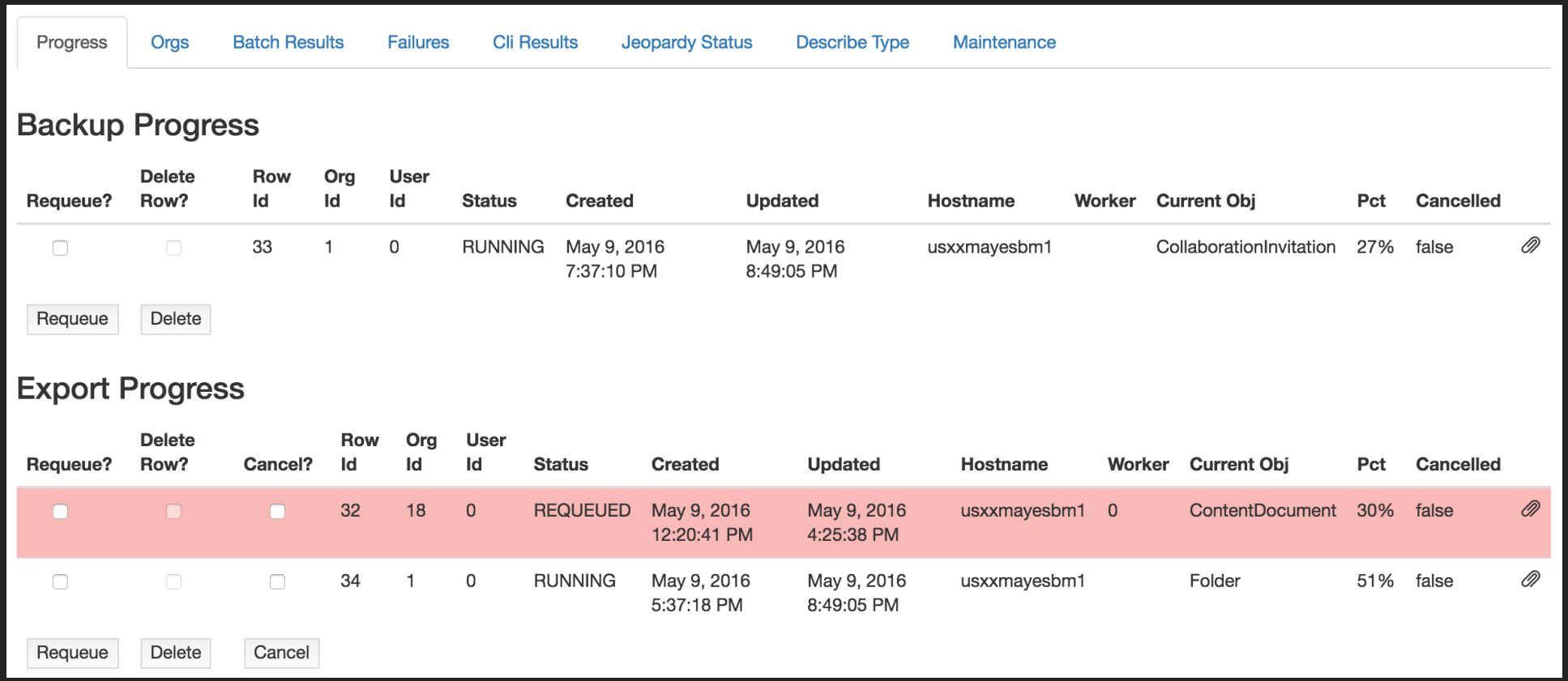
- Nagios for inspecting values and paging out on events:
  - CPU, memory, disk utilization, JVM heap usage, etc.
  - EC2 instance going down
  - Anything that has ever burned them in the past!
- Cloudwatch metrics going to graphite for now (InfluxDB soon)
- Pump out information to a Slack channel

INTERNAL API AND ADMIN INTERFACE

Helps with production support!

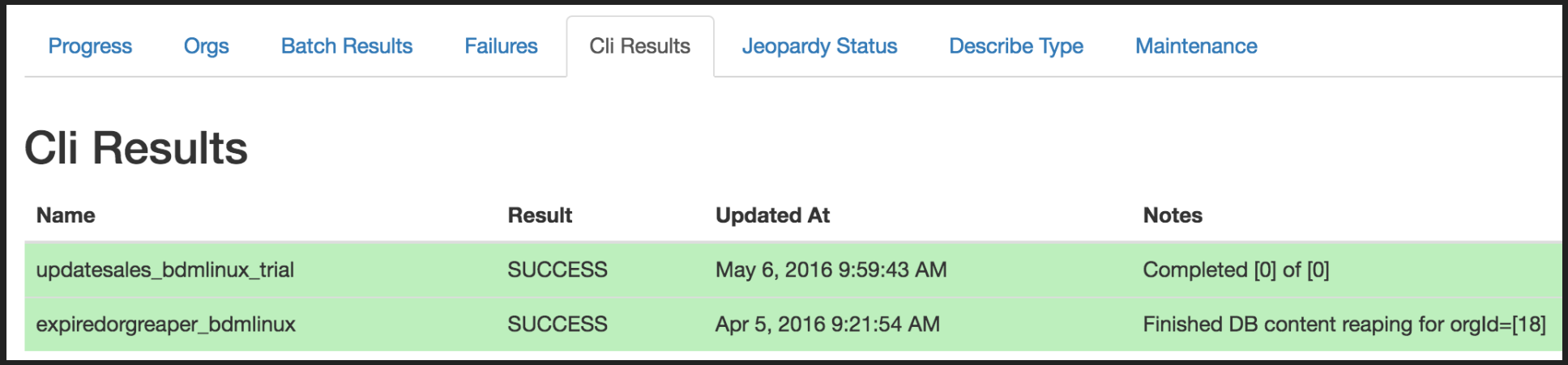
- Observe and intervene at runtime
- Provision new customers
- Some metrics/trends collection
- We can always improve it!

MONITORING PROGRESS



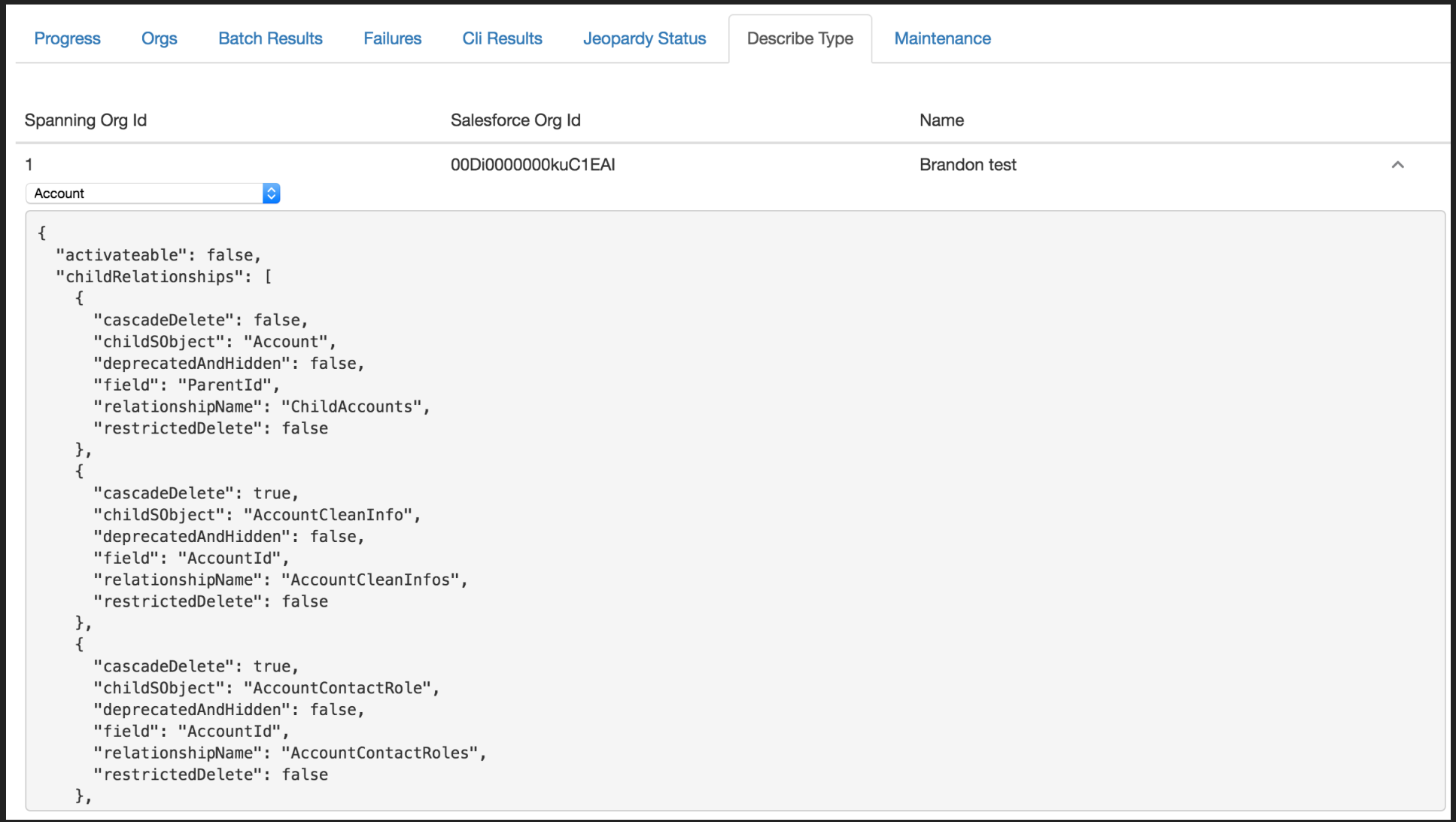
- Obviously this is internal only and doesn't have to be pretty...just functional.
- Gives supervisor a means to monitor the system without being at a shell (Mysql console, tailing logs)
- Helps enforce security and you can still have insight into the system without being part of the privileged roles

CLI RESULTS



Example: CRON jobs (running every N days or X minutes) that need to be monitored.

DESCRIBING OBJECTS



Example: Accessing org's structural info while not at shell!

This presentation still isn't over?

THE STRUGGLE IS REAL

- Speed at which dependencies change (and often include breaking changes)
  - npm shrinkwrap
  - Tests can and will save you
- Diagnosing node process crash and performance profiling
- Can't always test with "live" data or large enough data
  - Standup DB snapshot for testing, explain plans, etc.
  - Log too much at first, then decrease later after code is stable

TAKEAWAYS

- Build separate pieces that scale independently (SOA)
- Use queues/redis/whatever for interprocess communication
- Fail safely/silently/gracefully (build in fault tolerance, resumability, etc.)
- Choose persistence layer carefully but leave it open for change
- Create cool patterns and reuse them
- Be rigorous about your coding conventions and testing
- Be paranoid

QUESTIONS?

THANK YOU!

- on the twitters: @twofifty6
- slides posted to: github.com/bdmayes/NodeScaleMeetup
- more info: techblog.spanning.com