PRACTICAL PROJECTS WITH

WHO AM I?

Alex Sharp

Lead Developer at OptimisDev

@ajsharp (on the twitters)

alexjsharp.com (on the ‘tubes)

github.com/ajsharp (on the ‘hubs)

MAJOR POINTS

- Making the case for Mongo
- Brief intro to MongoDB
- Practical Projects

MAKING THE CASE

THE PROBLEMS OF SQL

A BRIEF HISTORY OF SQL

- Developed at IBM in early 1970's.
- Designed to manipulate and retrieve data stored in relational databases

A BRIEF HISTORY OF SQL

- Developed at IBM in early 1970's.
- Designed to manipulate and retrieve data stored in relational databases

this is a problem

WE DON'T WORK WITH DATA...

WE WORK WITH OBJECTS

WE DON’T CARE ABOUT

STORING DATA

WE CARE ABOUT

PERSISTING STATE

DATA != OBJECTS

THEREFORE...

BOLD STATEMENT FORTHCOMING...

relational DBs are an antiquated tool for our needs

OK, SO WHAT?

Relational schemas are designed to *store and query data*, not *persist objects*.

To reconcile this mismatch, we have ORM's

Object Relational

Mappers

ORMs enable us to create a *mapping* between our

native object model

and a *relational schema*

We're forced to create

relationships for *data*

when what we really want is

properties for *objects*.

THIS IS NOT EFFICIENT

A WEIRD EXAMPLE...

```
@alex = Person.new(
    :name => "alex",
    :stalkings => [
        Friend.new("Jim"),
        Friend.new("Bob")]
)
```

NATIVE RUBY OBJECT

```
<Person:0x10017d030 @name="alex",
 @stalkings=
  [#<Friend:0x10017d0a8 @name="Jim">,
   #<Friend:0x10017d058 @name="Bob">
]>
```

JSON REPRESENTATION

```
@alex.to_json
=> { name: "alex", stalkings: [{ name: "Jim" }, { name: "Bob" }] }
```

Relational Schema Representation

```yaml
people:
  - name

stalkings:
  - name
  - stalker_id
  - stalkee_id
```

SQL Schema Representation

```
people:
  - name

stalkings:
  - name
  - stalker_id
  - stalkee_id
```

What!?

Ruby -> JSON -> SQL

Ruby

```
<Person:0x10017d030 @name="alex",
@stalkings=
  [#<Friend:0x10017d0a8 @name="Jim">,
   #<Friend:0x10017d058 @name="Bob">
]>
```

JSON

```
@alex.to_json
{ name: "alex",
  stalkings: [{ name: "Jim" }, { name: "Bob" }]
}
```

SQL

```
people:
  - name

stalkings:
  - name
  - stalker_id
  - stalkee_id
```

Ruby -> JSON -> SQL

Ruby

```
<Person:0x10017d030 @name="alex",
@stalkings=
  [#<Friend:0x10017d0a8 @name="Jim">,
   #<Friend:0x10017d058 @name="Bob">
]>
```

JSON

```
@alex.to_json
{ name: "alex",
  stalkings: [{ name: "Jim" }, { name: "Bob" }]
}
```

SQL

```
people:
  - name

stalkings:
  - name
  - stalker_id
  - stalkee_id
```

Feels like we're working too hard here

You're probably thinking...

"SQL isn't *that* bad"

Maybe.

But we can do better.

THE IDEAL PERSISTENCE LAYER:

THE IDEAL PERSISTENCE LAYER:

1. To persist objects in their *native state*

THE IDEAL PERSISTENCE LAYER:

1. To persist objects in their *native state*
2. Does not interfere w/ application development

THE IDEAL PERSISTENCE LAYER:

1. To persist objects in their *native state*
2. Does not interfere w/ application development
3. Provides features needed to build modern web apps

MONGO TO THE RESCUE!

WHAT IS MONGODB?

mongodb is a high-performance, schema-less, scalable, document-oriented database

STRENGTHS

SCHEMA-LESS

SCHEMA-LESS

- Great for rapid, iterative, agile development

SCHEMA-LESS

- Great for rapid, iterative, agile development
- Makes things possible that in an RDBMS are either

  a. impossibly hard

  b. virtually impossible

  c. way harder than they should be

DOCUMENT-ORIENTED

DOCUMENT-ORIENTED

- Mongo stores *documents*, not rows
  - documents are stored as binary json

DOCUMENT-ORIENTED

- Mongo stores *documents*, not rows
  - documents are stored as binary json
- Rich, familiar query syntax

DOCUMENT-ORIENTED

- Mongo stores *documents*, not rows
  - documents are stored as binary json
- Rich, familiar query syntax
- Embedded documents eliminate many modeling headaches

QUERY EXAMPLES

sample mongo document

```
{'status': 500,
 'request_method': 'post',
 'controller_action': 'notes#create',
 'user':
  {'first_name': 'John',
   'last_name': 'Doe'}
}
```

QUERY EXAMPLES

```
logs.find('status' => 500)

# SQL equivalent:
# select * from logs WHERE logs.status = 500
```

```
{'status': 500,
 'request_method': 'post',
 'controller_action': 'notes#create',
 'user':
   {'first_name': 'John',
    'last_name': 'Doe'}
}
```

QUERY EXAMPLES

```
logs.find('status' => 500, 'user.last_name' => 'Doe')

# SQL equivalent: none
```

```
{'status': 500,
 'request_method': 'post',
 'controller_action': 'notes#create',
 'user':
   {'first_name': 'John',
    'last_name': 'Doe'}
}
```

QUERY EXAMPLES

```
logs.find('status' => 500, 'user.last_name' => 'Doe')
```

```
{'status': 500,
 'request_method': 'post',
 'controller_action': 'notes#create',
 'user':
   {'first_name': 'John',
    'last_name': 'Doe'}
}
```

user is an embedded document

QUERY EXAMPLES

```
logs.find('status' => 500, 'user.last_name' => 'Doe')
```

in sql, we would have JOIN-ed to the 'users' table to do this query

QUERY EXAMPLES

```
# also works w/ regex
logs.find('request_method' => /p/i)

# SQL equivalent:
# select * from logs where request_method LIKE %p%

  {'status': 500,
   'request_method': 'post',
   'controller_action': 'notes#create',
   'user':
     {'first_name': 'John',
      'last_name': 'Doe'}
  }
```

BIG DATA

BIG DATA

- Built to scale horizontally into the cloudz

BIG DATA

- Built to scale horizontally into the cloudz
- Auto-sharding
  - set a shard-key, a cluster of nodes, go

FAST WRITES

FAST WRITES

- In-place updates
  - i.e. *upsert*

FAST WRITES

- In-place updates
  - i.e. *upsert*
- Lot's of db commands for fast updates
  - $set, $unset, $inc, $push

UPSERT

- Update if present, insert otherwise
  - careful: update does a hard replace of the entire document

```
# initial insert
posts.save(:title => 'my post', :body => '...')

# update-in-place "upsert"
posts.save(:_id => 'post_id', :title => 'new title', :body => '...')
```

AGGREGATION

Map/reduce for aggregation

AGGREGATION

```
// map
function() {
  emit(this.controller_action, { count: 1 });
}

// reduce
function(key, values) {
  var sum = 0;
  values.forEach(function(doc) {
    sum += doc.count;
  });

  return { count: sum };
}
```

FAST READS

Optimize reads with indexes, just like an rdbms

```
# index is only created if it does not exist
logs.create_index('status')

# can index on embedded documents
logs.create_index('user.first_name')

# create compound indexes
logs.create_index(
  [['user.first_name', 1],
   ['user.last_name',  1]]
)
```

WEAKNESSES

JOINS

nope.

MULTI-DOCUMENT TRANSACTIONS

no-sir-ee.

RELATIONAL INTEGRITY

not a chance.

PRACTICAL PROJECTS

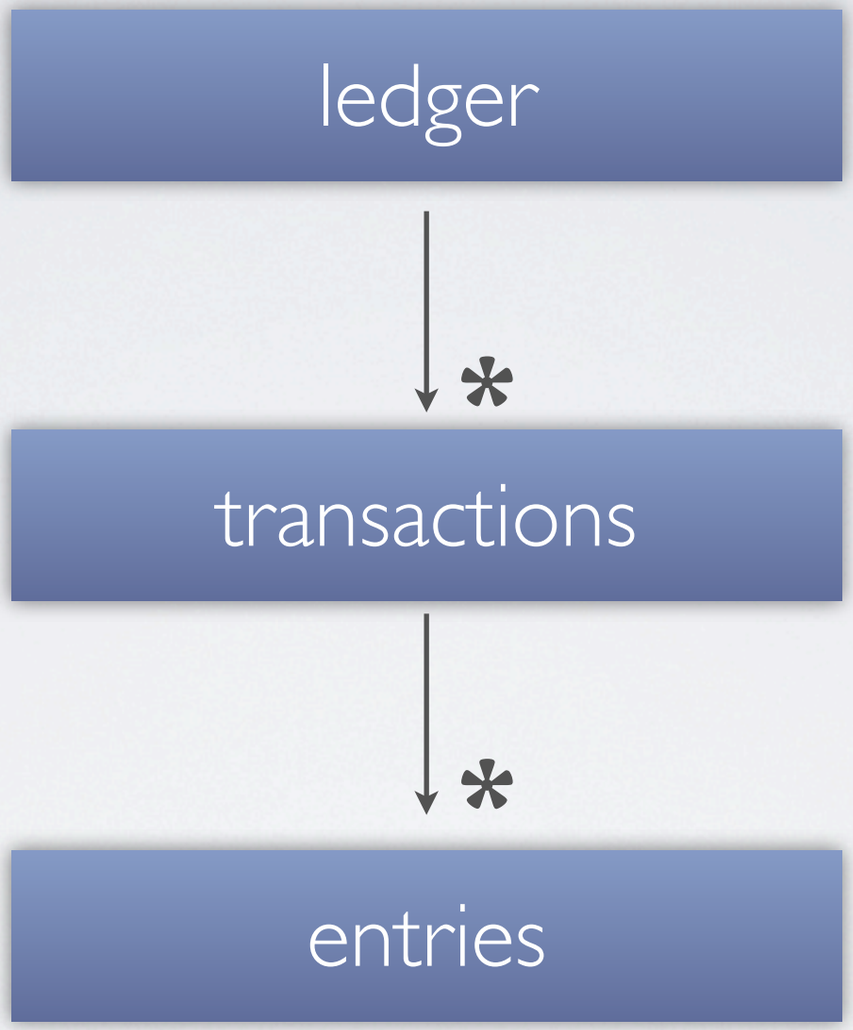
1. Accounting Application

2. Capped Collection Logging

3. Blogging app

GENERAL LEDGER ACCOUNTING APPLICATION

THE OBJECT MODEL



THE OBJECT MODEL

	#	Credits	Debits
Line item	1	{ :account => “Cash”, :amount => 100.00 }	{ :account => “Notes Pay.”, :amount => 100.00 }
Line item	2	{ :account => “A/R”, :amount => 25.00 }	{ :account => “Gross Revenue”, :amount => 25.00 }

Ledger Entries

THE OBJECT MODEL

THE OBJECT MODEL

- Each ledger *line item* belongs to a *ledger*

THE OBJECT MODEL

- Each ledger *line item* belongs to a *ledger*
- Each ledger *line item* has two (or more) ledger *entries*
  - must have at least one credit and one debit
  - credits and debits must balance

THE OBJECT MODEL

- Each ledger *line item* belongs to a *ledger*
- Each ledger *line item* has two (or more) ledger *entries*
  - must have at least one credit and one debit
  - credits and debits must balance
- These objects must be transactional

SQL-BASED OBJECT MODEL

```
@debit  = Entry.new(:account => 'Cash',
  :amount => 100.00, :type => 'credit')

@credit = Entry.new(:account => 'Notes Pay.',
  :amount => 100.00, :type => 'debit')


@line_item = LineItem.new(:ledger_id => 1,
  :entries => [@debit, @credit])
```

SQL-BASED OBJECT MODEL

```
@debit  = Entry.new(:account => 'Cash',
  :amount => 100.00, :type => 'credit')

@credit = Entry.new(:account => 'Notes Pay.',
  :amount => 100.00, :type => 'debit')

@line_item = LineItem.new :ledger_id => 1, :entries => [@debit, @credit]
```

In a relational schema, we need a database transaction to ensure multi-row atomicity

SQL-BASED OBJECT MODEL

```
@debit  = Entry.new(:account => 'Cash',
  :amount => 100.00, :type => 'credit')

@credit = Entry.new(:account => 'Notes Pay.',
  :amount => 100.00, :type => 'debit')

@line_item = LineItem.new :ledger_id => 1, :entries => [@debit, @credit]
```

Because we need to create 3 new records in the database here

MONGO-FIED MODELING

```
@line_item = LineItem.new(:ledger_id => 1,
  :entries => [
    {:account => 'Cash',        :type => 'debit', :amount => 100},
    {:account => 'Notes pay.', :type => 'debit', :amount => 100}
  ]
)
```

MONGO-FIED MODELING

```
@line_item = LineItem.new(:ledger_id => 1,
  :entries => [
    {:account => 'Cash',        :type => 'debit', :amount => 100},
    {:account => 'Notes pay.', :type => 'debit', :amount => 100}
  ]
)
```

This is the perfect case for embedded documents.

MONGO-FIED MODELING

```
@line_item = LineItem.new(:ledger_id => 1,
  :entries => [
    {:account => 'Cash',        :type => 'debit', :amount => 100},
    {:account => 'Notes pay.', :type => 'debit', :amount => 100}
  ]
)
```

We will never have more than a few entries (usually two)

MONGO-FIED MODELING

```
@line_item = LineItem.new(:ledger_id => 1,
  :entries => [
    {:account => 'Cash',        :type => 'debit', :amount => 100},
    {:account => 'Notes pay.', :type => 'debit', :amount => 100}
  ]
)
```

Embedded docs are perfect for "contains" many relationships

MONGO-FIED MODELING

```
@line_item = LineItem.new(:ledger_id => 1,
  :entries => [
    {:account => 'Cash',        :type => 'debit', :amount => 100},
    {:account => 'Notes pay.', :type => 'debit', :amount => 100}
  ]
)
```

DB-level transactions no-longer needed.
Mongo has single document atomicity.

WINS

WINS

- Simplified, de-normalized data model
  - Objects modeled differently in Mongo than SQL

WINS

- Simplified, de-normalized data model
  - Objects modeled differently in Mongo than SQL
- Simpler data model + no transactions = WIN

LOGGING WITH CAPPED COLLECTIONS

BASELESS STATISTIC

95% of the time logs are NEVER used.

MOST LOGS ARE ONLY USED WHEN SOMETHING GOES WRONG

CAPPED COLLECTIONS

- Fixed-sized, limited operation, auto age-out collections (kinda like memcached, except persistent)
- Fixed insertion order
- Super fast (faster than normal writes)
- Ideal for logging and caching

GREAT. WHAT'S THAT MEAN?

We can log additional pieces of arbitrary data effortlessly due to Mongo's schemaless nature.

THIS IS AWESOME

CAPTURE.
QUERY.
ANALYZE.
PROFIT.

ALSO A REALLY HANDY TROUBLESHOOTING TOOL

User-reported errors are difficult to diagnose

Wouldn’t it be useful to see the complete click-path?

BUNYAN

http://github.com/ajsharp/bunyan

Thin ruby layer around a MongoDB capped collection

BUNYAN

http://github.com/ajsharp/bunyan

Still needs a middleware api.

Want to contribute?

```ruby
require 'bunyan'

Bunyan::Logger.configure do
  database   'my_bunyan_db'
  collection "#{Rails.env}_bunyan_log"
  size       1073741824 # 1.gigabyte
end
```

PAPERMILL

http://github.com/ajsharp/papermill

SINATRA FRONT-END TO BUNYAN

PAPERMILL

http://github.com/ajsharp/papermill

DEMO TIME

12 Standard Screen Patterns | Why aren't you using git-flow | Chain Constructors | Jasmine: Javascript Testing | git tpope's vim-rake at master | git lib/mongo/collection.rb at | git paulelliott's fabrication at

localhost:3001/#

Bills | Aho/Ullman Foundat | An Introduction to D | Lessons Learned: Wh | 10 Things I Learned | Giles Bowkett: Two P | Aaron Quint – Makin

Papermill

Search Parameters

Email remove

add

Search

BLOGGING APPLICATION

BLOGGING APPLICATION

Much easier to model with Mongo than a relational database

BLOGGING APPLICATION

A post has an author

BLOGGING APPLICATION

A post has an author

A post has many tags

BLOGGING APPLICATION

A post has an author

A post has many tags

A post has many comments

BLOGGING APPLICATION

A post has an author

A post has many tags

A post has many comments

Instead of joining separate tables,
we can use embedded documents.

MONGOMAPPER

MONGOMAPPER

- MongoDB "ORM" developed by John Nunemaker

MONGOMAPPER

- MongoDB "ORM" developed by John Nunemaker
- Very similar syntax to DataMapper

MONGOMAPPER

- MongoDB "ORM" developed by John Nunemaker
- Very similar syntax to DataMapper
  - Declarative rather than inheritance-based

MONGOMAPPER

- MongoDB "ORM" developed by John Nunemaker
- Very similar syntax to DataMapper
  - Declarative rather than inheritance-based
- Very easy to drop into rails

MONGOMAPPER

```
require 'mongo'
connection = Mongo::Connection.new.db("blog_app")

class Post
  include MongoMapper::Document

  belongs_to :author, :class_name => "User"

  key :title,        String,  :required => true
  key :body,         String,  :required => true
  key :author_id,    Integer, :required => true
  key :published_at, Time
  key :published,    Boolean, :default => false
  key :tags,         Array,   :default => []
  timestamps!
end
```

INDEXES!

```
Post.collection.create_index('tags')

Post.collection.create_index('title')
```

WINS

WINS

Simpler persistence model matches simple object model

NEXT STEPS

OBJECT-DOCUMENT MAPPERS

- Ruby mongo driver
  - github.com/mongodb/mongo-ruby-driver
- MongoMapper
  - github.com/jnunemaker/mongomapper
- Mongoid
  - github.com/durran/mongoid

MONGOMAPPER

- DataMapper-like API
- good for moving from a sql schema to mongo
- easy to drop into rails
- works with rails 3

MONGOID

- DataMapper-like API
- uses ActiveModel, so familiar validation syntax
- more opinionated towards embedded docs
- easy to drop into rails
- rails 2 - mongoid 1.x
- rails 3 - mongoid 2.x

mongodb.org

MOST IMPORTANTLY

最も重要な...

WE MUST ARRIVE AT THE WHISKEY BAR EARLIER TOMORROW

我々はウイスキーバーは明日、以前に到着する必要があります

BECAUSE IT CLOSES AT MIDNIGHT

;-)

それは真夜中に閉じるため

QUESTIONS?

THANKS!