PRACTICAL

PROJECTS WITH

mongoDB

WHO AM I?

Alex Sharp

Lead Developer at OptimisDev

@ajsharp (on the twitters)

alexjsharp.com (on the 'tubes)

github.com/ajsharp (on the 'hubs)

OptimisDev®

MAJOR POINTS

- Brief intro to MongoDB
- Practical Projects
- Making the case for Mongo (time permitting)

BRIEF INTRO

WHAT IS MONGODB?

mongodb is a high-performance, schema-less, 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
- Allows for a rich query syntax

DOCUMENT-ORIENTED

- Mongo stores *documents*, not rows
  - documents are stored as binary json
- Allows for a rich query syntax
- Embedded documents eliminate many modeling headaches

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
- still in alpha, but production-ready soon

FAST WRITES

FAST WRITES

- Mongo writes are "fire-and-forget"
  - you can get a response with `getLastError()`

FAST WRITES

- Mongo writes are "fire-and-forget"
  - you can get a response with `getLastError()`
- In-place updates

FAST WRITES

- Mongo writes are "fire-and-forget"
  - you can get a response with `getLastError()`
- In-place updates
- Lot's of db commands for fast updates

AGGREGATION

Map/reduce for aggregation

READS FAST TOO

Optimize reads with indexes, just like an rdbms

WEAKNESSES

JOINS

nope.

MULTI-DOCUMENT TRANSACTIONS

no-sir-ee.

RELATIONAL INTEGRITY

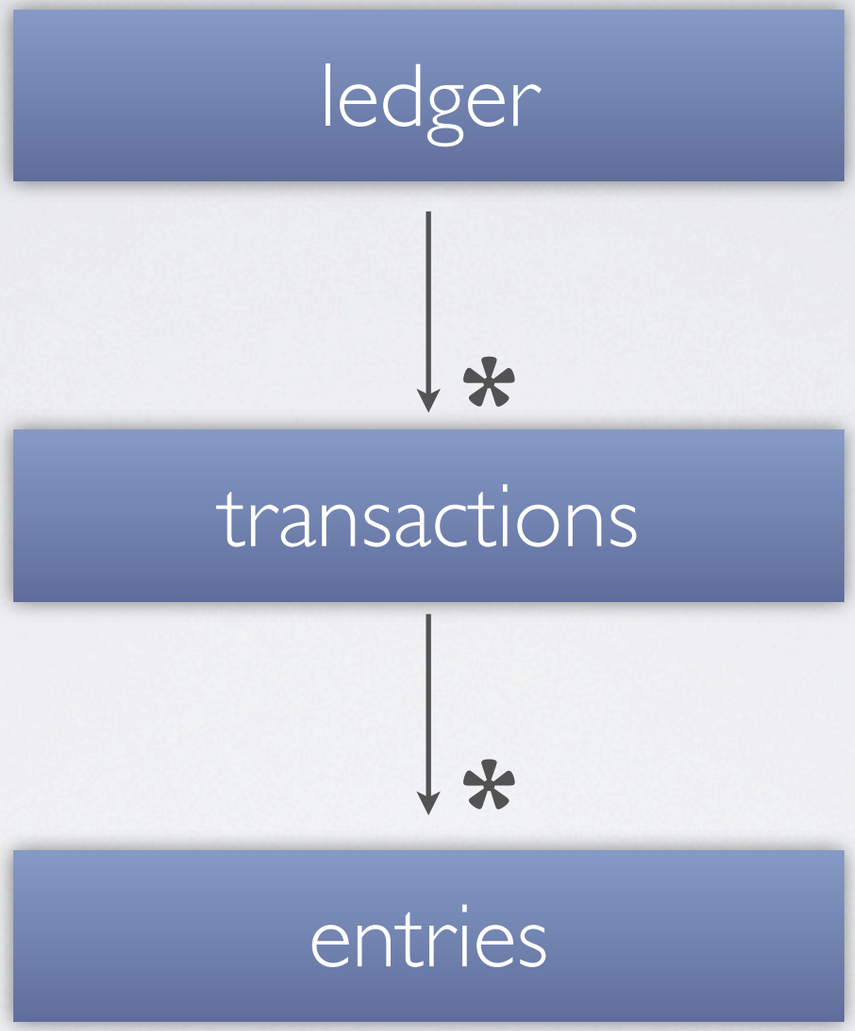
not a chance.

PRACTICAL PROJECTS

1. Accounting Application
2. Capped Collection Logging
3. Blogging app
4. Reporting 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 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 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 SQL 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]
```

Remember: we have 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.

LESSONS

LESSONS

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

LESSONS

- 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

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 |c|
  c.database   'my_bunyan_db'
  c.collection "#{Rails.env}_bunyan_log"
  c.size 1073741824 # 1.gigabyte
end
```

PAPERMILL

http://github.com/ajsharp/papermill

SINATRA FRONT-END TO BUNYAN

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.

OptimisDev®

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

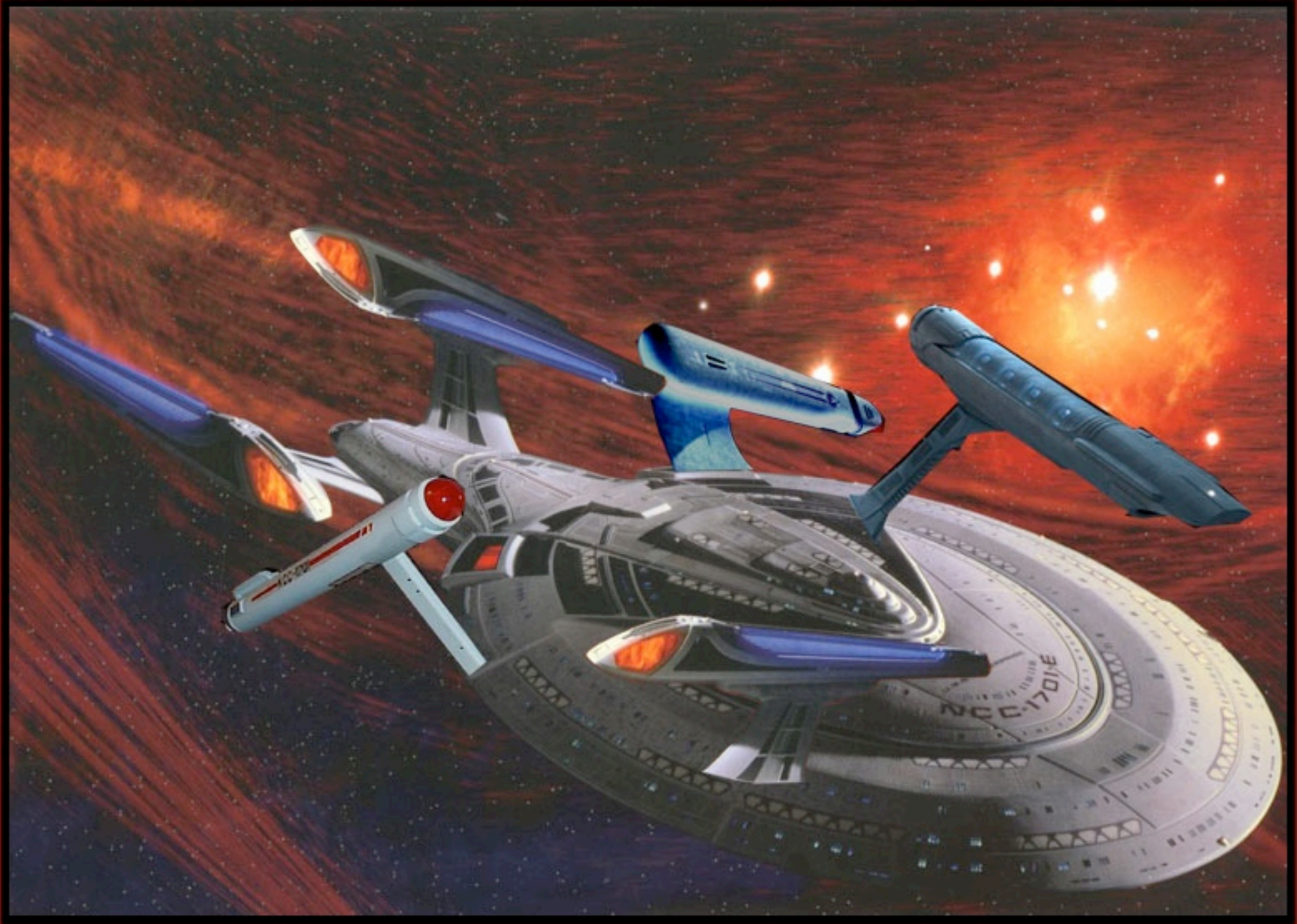
```
require 'mongo'
connection = Mongo::Connection.new.db("#{Rails.env}_app")

class Post
  include MongoMapper::Document

  belongs_to :author, :class_name => "User"

  key :title,        String,  :reuqired => 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
```

REPORTING APP



ENTERPRISE SOFTWARE

It's really complicated, it took forever to build, and it cost a whole lot.

*It **must** be better.*

Andrew Burke

www.shindigital.com

REPORTING IS HARD

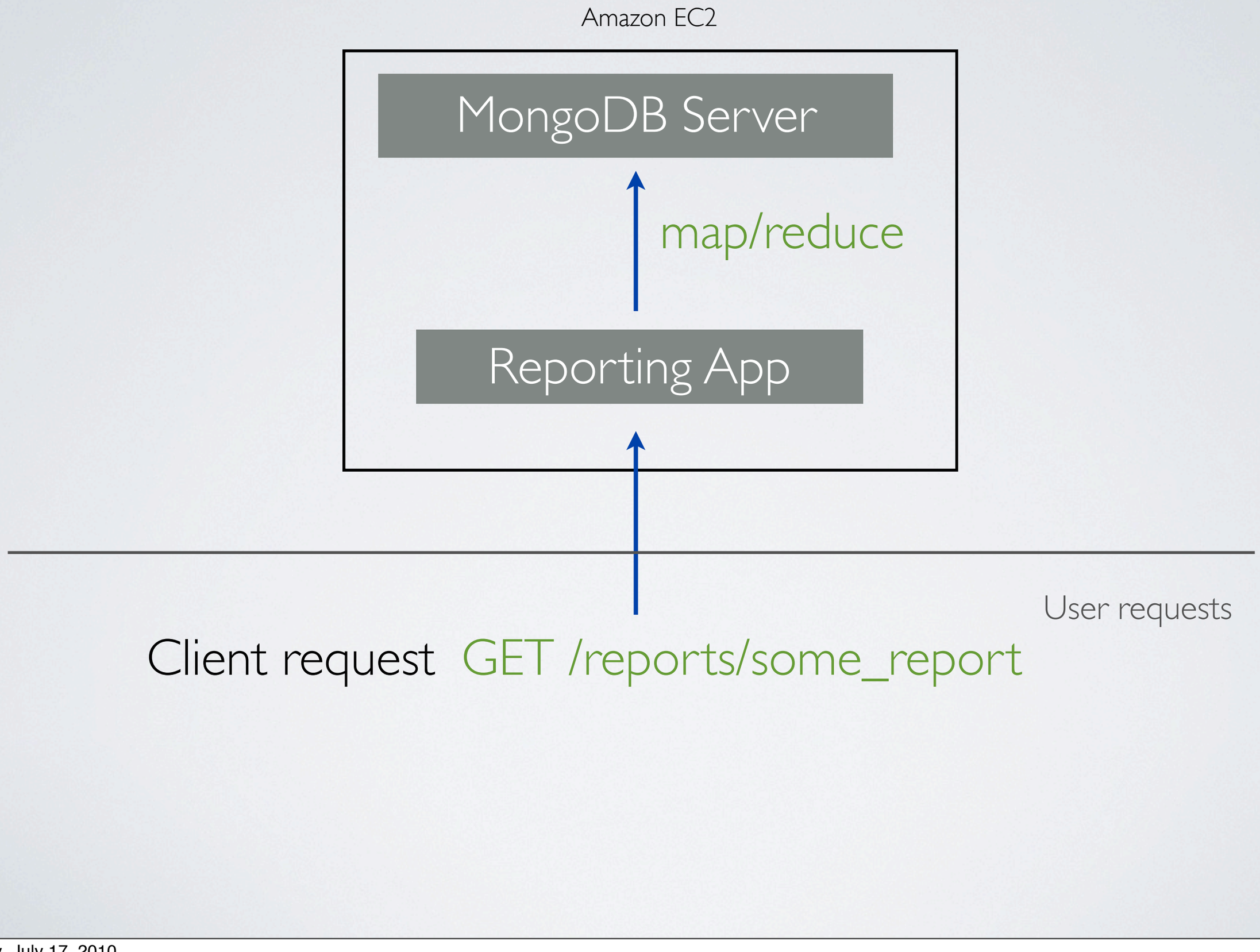
- Synchronization
  - Data, application API, schema
- Schema synchronization is the hard one

WHY MONGO?

- Aggregation support with Mongo's map/reduce support
- Mongo is good (and getting better) at handling big data sets
- Schemaless is perfect for a flattened data set


Amazon EC2
MongoDB Server
Reporting App
WAN
Rackspace
Transform
Main App
MySQL

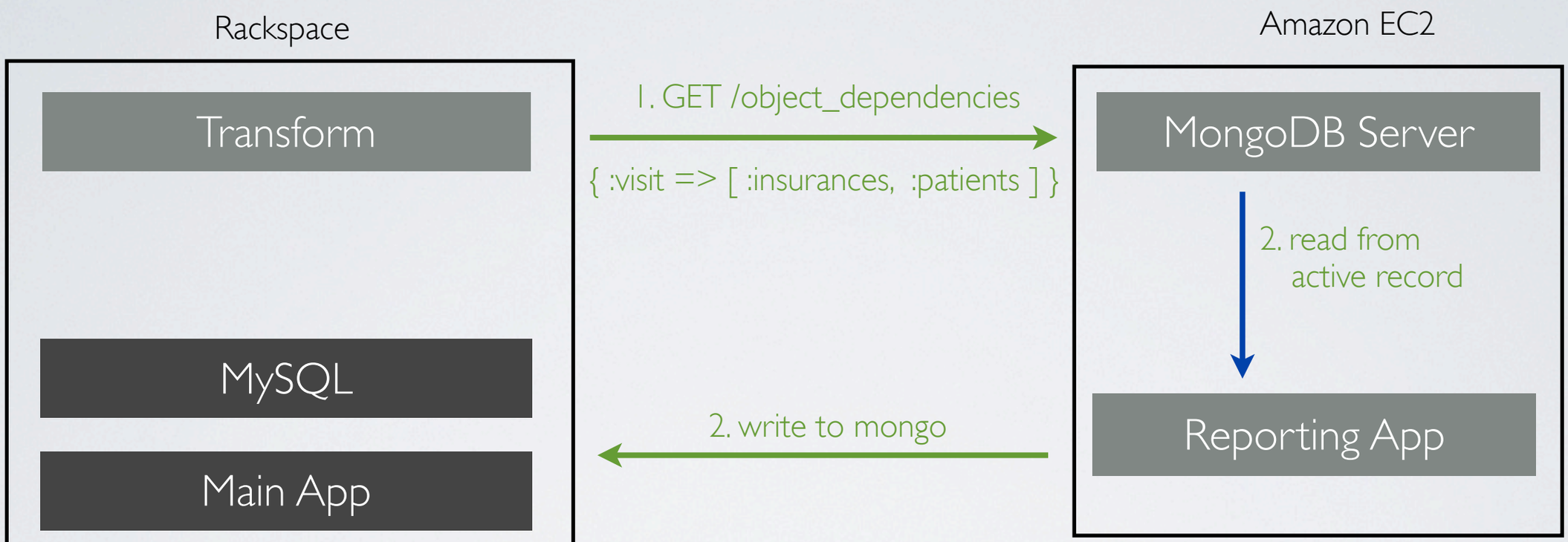
CLIENT REQUESTS



TRANFORMATION

Data needs to *extracted* from the the main app, *transormed* (i.e. flattened) into some other form, and loaded into the reporting app for aggregation

TRANSFORMATION



MAKING THE CASE

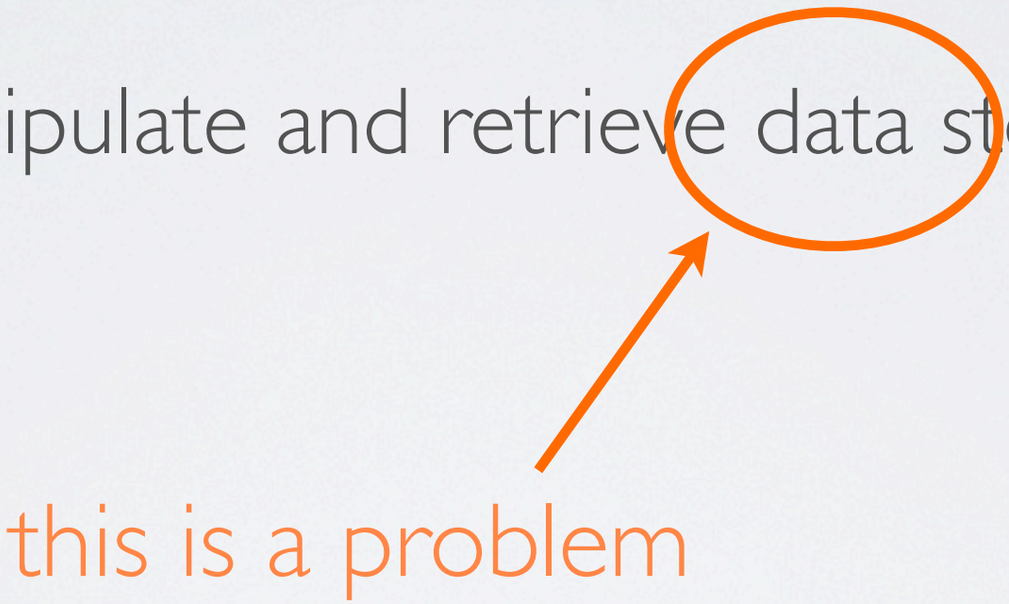
THE PROBLEMS OF SQL

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



WHY??

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...

RELATIONAL DBS ARE AN ANTIQUATED TOOL FOR OUR NEEDS

OK, SO WHAT?

SQL schemas are designed for storing and querying *data*, not *persisting objects*.

To reconcile this mismatch, we have ORM's

Object Relational *Mappers*

OK, SO WHAT?

We need ORMs to bridge the gap (map) between our native Ruby object model and a SQL schema

We're forced to create *relationships* for *data*

when what we really want is *properties* for our *objects*.

THIS IS NOT EFFICIENT

```
@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/Mongo Representation

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

SQL Schema Representation

```
people:
  - name

stalkings:
  - name
  - stalker_id
```

SQL Schema Representation

```
people:
  - name
```

```
stalkings:
  - name
  - stalker_id
```

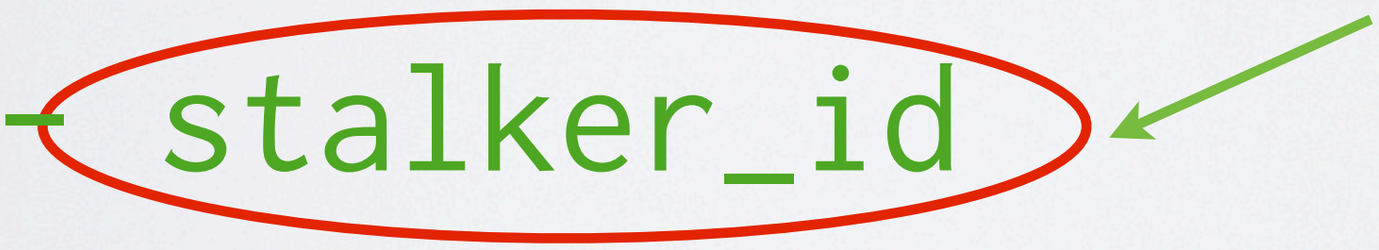
What!?

SQL Schema Representation

```
people:
    - name
```

```
stalkings:
    - name
    - stalker_id
```

what's a stalker_id?

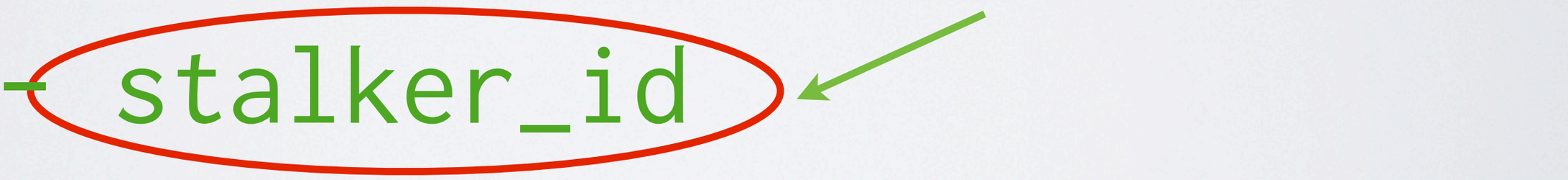


SQL Schema Representation

```
people:
    - name
```

```
stalkings:
    - name
    - stalker_id
```

this is our sql mapping



SQL Schema Representation

```
people:
    - name
```

```
stalkings:
    - name
    - stalker_id
```

this is the
“pointless” join

Ruby -> JSON -> SQL

Ruby

JSON

SQL

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

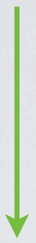
```
@alex.to_json
{ name: "alex",
  stalkings: [{ name: "Jim" }, { name: "Bob" }]
}
```

```
people:
  - name

stalkings:
  - name
  - stalker_id
```

Ruby -> JSON -> SQL

Ruby



JSON

SQL

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

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

```
people:
  - name

stalkings:
  - name
  - stalker_id
```

Feels like we're having to work too hard here

You're probably thinking...

"Listen GUY, SQL isn't *that* bad"

Maybe.

But we can do much, much better.

NEEDS FOR A PERSISTENCE LAYER:

NEEDS FOR A PERSISTENCE LAYER:

1. To persist the native state of our objects

NEEDS FOR A PERSISTENCE LAYER:

1. To persist the native state of our objects

2. Should NOT interfere w/ application development

NEEDS FOR A PERSISTENCE LAYER:

1. To persist the native state of our objects

2. Should NOT interfere w/ application development

3. Provides features necessary to build modern web apps

NEXT STEPS

USING MONGO W/ RUBY

- Ruby mongo driver
- MongoMapper (github.com/jnunemaker/mongomapper)
- MongoID (github.com/durran/mongoid)
- Many other ORM/ODM's
  - moneta (github.com/wycats/moneta)

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 embedded docs
- easy to drop into rails
- rails 2 - mongoid 1.x
- rails 3 - mongoid 2.x

QUESTIONS?

THANKS!