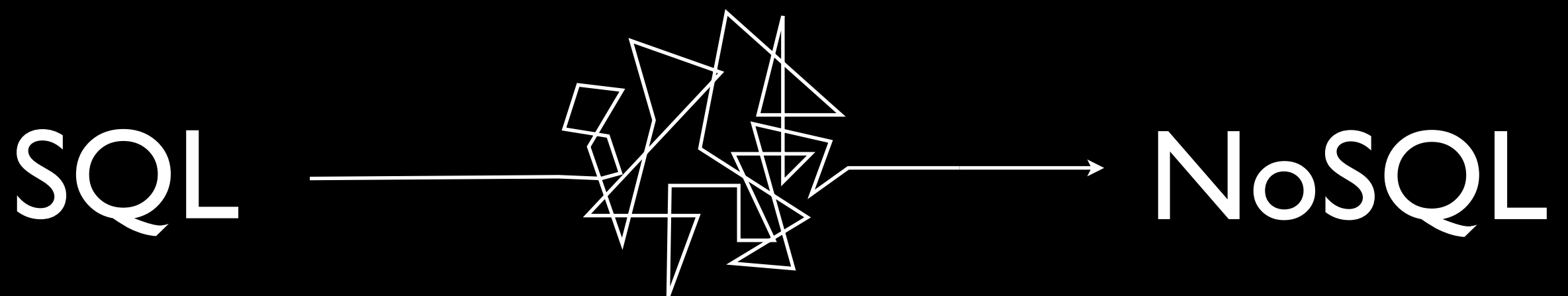


Denormalizing Your Rails Application

danlucraft

Ruby Manor 2: Manor Harder / 12 December 2009



You're on the left. You've all heard how great it is on the right. But getting there... not so clear. How do you introduce elements of NoSQL into a pure SQL Rails project.

Faced with this problem, I hit the blogs. Didn't find much. Here's what we did instead.



This is where I work. We're a startup based at Old Street, and www.songkick.com is a social network based around live music, and we aggregate ticket information to make sure our users never miss gigs.

Home

London events (Change city)

What is Songkick?

Sign up

Muse

Track 5388 people are tracking this artist.

Sign up now and never miss a gig again.
Track artists. Get email alerts. Buy tickets.



29 upcoming concerts **880** past concerts in 333 cities

See full gigography (880 past concerts)
All: Photos (225) Videos (12) Posters (38)



Flag a problem

Muse is playing near London! Not in London?

Upcoming concerts

See all

Friday 10 September 2010

Add an event

Muse

Wembley Stadium
London, UK

8 people

Buy tickets

You're looking for events near

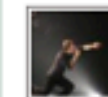
London, UK (Change)

Similar artists on tour

powered by **lost.fm**



Welcome
1 upcoming and 5 past concerts



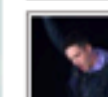
Coldplay
12 upcoming and 674 past concerts



Arctic Monkeys
16 upcoming and 397 past concerts



Franz Ferdinand
5 upcoming and 490 past concerts



The Killers
4 upcoming and 717 past concerts



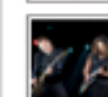
Green Day
9 upcoming and 597 past concerts



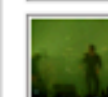
Snow Patrol
3 upcoming and 585 past concerts



Placebo
8 upcoming and 804 past concerts



Metallica
29 upcoming and 1588 past concerts



My Chemical Romance
2 upcoming and 287 past concerts

Highest concert attendance



maysiefurs
16 times



nickpalaceh
14 times

We have 500,000 artists in our database. This is one of them. There are lots of interesting numbers on this page. Some of them are costly to compute.

Home London events (Change city) What is Songkick?

Muse

Track 5388 people are tracking this artist.

Sign up now and never miss a gig again.
Track artists. Get email alerts. Buy tickets.

ON TOUR 29 upcoming concerts 880 past concerts in 333 cities

See full gigography (880 past concerts)
All: Photos (225) Videos (12) Posters (38)

Flag a problem



Muse is playing near London! Not in London?

Upcoming concerts

Friday 10 September 2010

Muse





















Wembley Stadium
London, UK

8 people

Add an event

Buy tickets

Highest concert attendance

 maysiefurs 16 times	 nickpalaceh 14 times
 Louatiqua 13 times	 laurabrooks 12 times
 el_nij 12 times	 davepickett82 10 times
 Maxnot 9 times	 Blonde 8 times
 hyperbaby 8 times	 mannyer 7 times
 anthonyherron 7 times	 debrarr 7 times
 sylvya 7 times	 giollaiosa 6 times
 luckysalt 6 times	 Lyakali 6 times
 threesunrises 6 times	 il_sharky 6 times
 wemster 6 times	 peterrooney 6 times

See all 734 people

Highest concert attendance

 maysiefurs 16 times	 nickpalaceh 14 times
---	--





















This module shows which of Songkick's users have seen Muse live the most.

This is only one example. Our data is VERY relational and there are a ton of different places we do this kind of thing.

big slow join

~~fragment caching~~

~~memcached~~

Highest concert attendance	
 maysiefurs 16 times	 nickpalaceh 14 times
 Louatiqua 13 times	 laurabrooks 12 times
 el_nij 12 times	 davepickett82 10 times
 Maxnot 9 times	 Blonde 8 times
 hyperbaby 8 times	 mannyer 7 times
 anthonyherron 7 times	 debrarr 7 times
 sylvya 7 times	 giollaiosa 6 times
 luckysalt 6 times	 Lyakali 6 times
 threesunrises 6 times	 il_sharky 6 times
 wemster 6 times	 peterrooney 6 times
See all 734 people	

Big slow join. This joins over 5 tables: Users–UserAttendances–Events–Performances–Artists. None of those has fewer than 100k rows, some have many millions.

Why is fragment caching not great for us?

- No good for Google to generate for the *next* user. We want SEO!
- Hard to pre-generate without using your entire Rails stack.
- Hard to change styles and regenerate HTML for 500,000 artists.
- 6k HTML per artist == a lot of RAM! Especially when you consider that the slow part is generating just 41 integers that make up the information.
- Sure fragment caching makes caching easy, but that isn't so hard anyway, the trick is expiration.

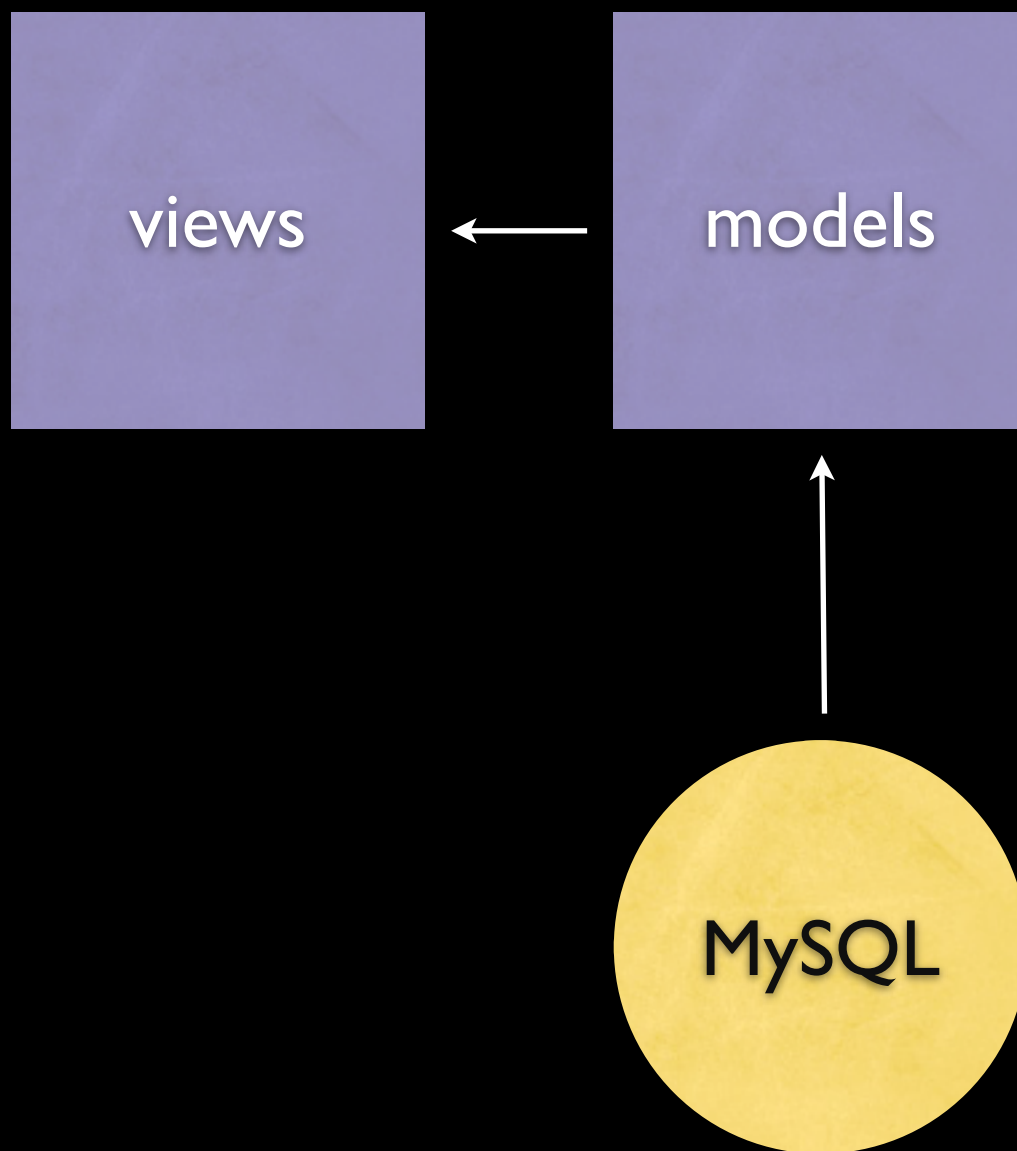
Why is memcache not great for us?

- Same reasons
- We want persistence. Would take days to regenerate all our data, in the mean time the site is slow.
- Can't be sure the data is available, what with expiration.
- RAM limited.
- Why **not** use something else?

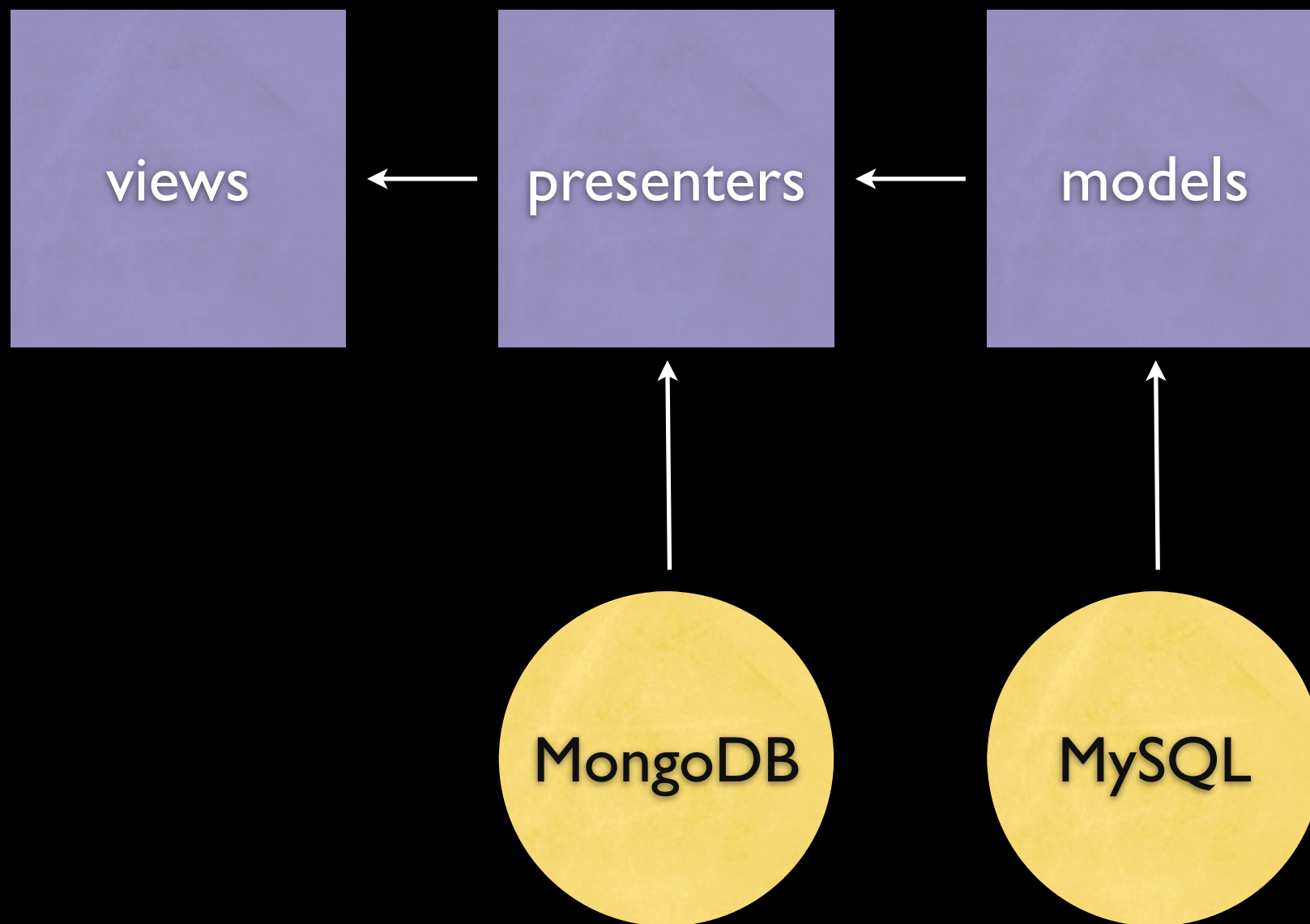


Why do we like Mongo?

- Used in for internal admin apps – works great. (We’ve also experimented a bit with CouchDB and Redis.)
- Schema-less which is great for our denormalized data which is changing a lot. (Schema less databases are a great fit with dynamic languages.)
- Pretty quick. Stores most/all of our db in RAM.
- Supports sharding (or close to supporting it anyway).
- Seems more mature than some....
- Fully supported Ruby driver. (With responsive IRC and developers.)



This is our previous setup. The views (or controllers) talk to the models which pull the data from MySQL.

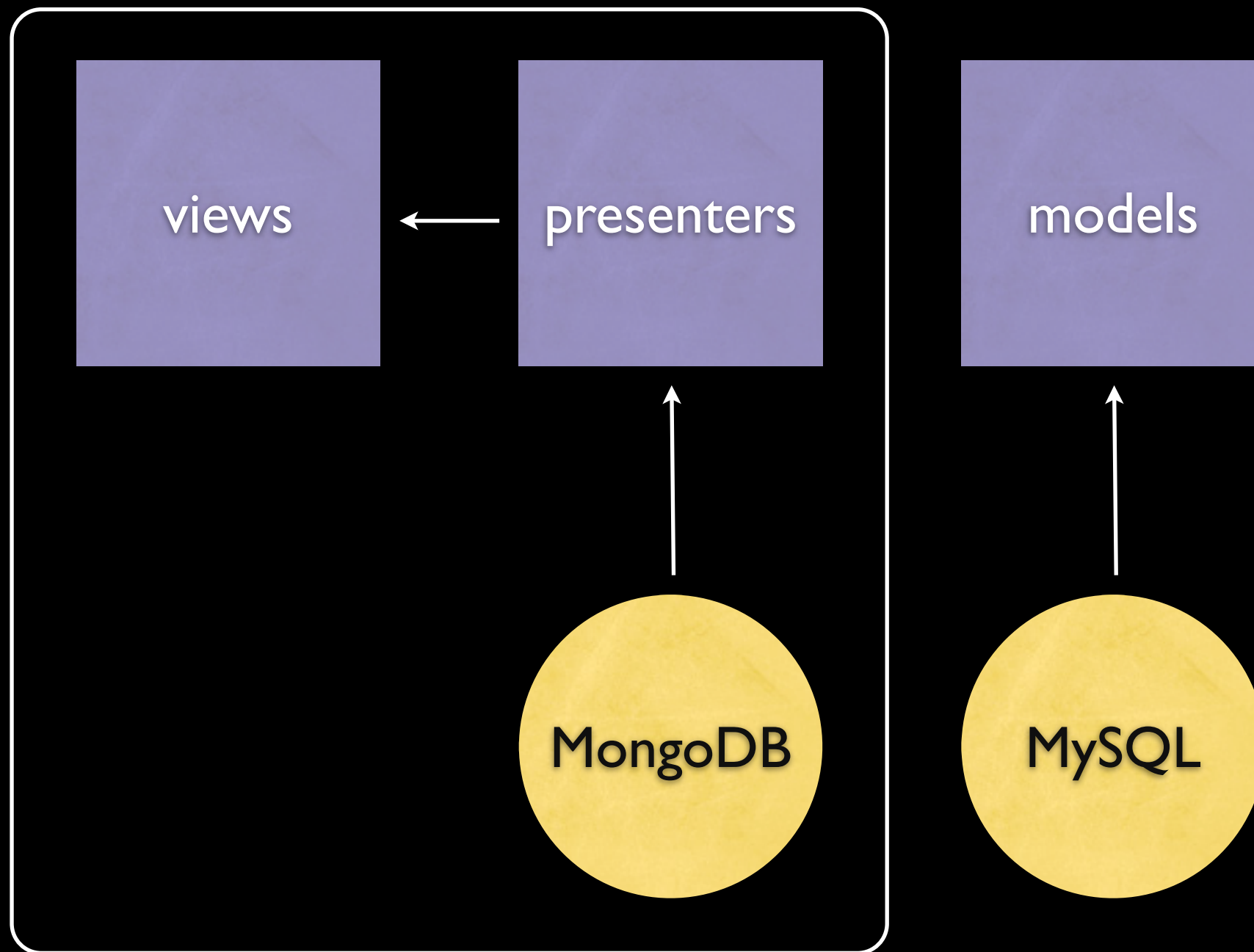


This is what we are moving to. There is a presentation layer in-between the views and the models that pulls information from MongoDB by preference, and only talks to the models if it has to.

We've been using presenters in our app for a long time. Extra classes are essential for managing complexity on large pages. But we've never been totally clear on their exact scope. Are they attached to views or models? Do they talk to the database or do they just wrap models and turn them into strings? Do we want one object for a page with mixed in modules for each subsection, or do we want lots of smaller objects for each subsection.

We've had all of these patterns in our code as we have experimented, and we have not come to a conclusion about the right responsibilities of the Presenters. Now we are much clearer. Presenters marshal data and store it in our Mongo database. Answers to all the questions drop out and it becomes obvious that there's only one way to write the class. This gives the strong impression that we are on the right track.

Website



This is perhaps our long-term destination. Mongo is suitable as a general purpose database so there's no reason that the website has to talk to MySQL at all, at least for read operations. Presenters can become first class objects.

We can do this incrementally, where it makes sense. And we'll always keep the SQL database because our data is incredibly relational.

Let's show how this is implemented for the module we saw earlier.

```
module Presenters::Artists
  class AttendancePresenter

    def initialize(artist)
      @artist = artist
    end

    def total_num_users_seen_live
      @artist.total_num_users_seen_live
    end

    def top_users_seen_live
      @artist.top_users_seen_live
    end

  end
end
```

This is an example of the AttendancePresenter for the Artist page. This is close to being real code. There is one presenter per artist page, therefore the presenter wraps an artist model.

Notice that the interface to the presenter and to the model is the same. The views don't care which is passed to them, which keeps us flexible.

This presenter pulls out a couple of pieces of information that are each slow to calculate. We want to store those pieces in the Silo ...

```
module Presenters::Artists
  class AttendancePresenter

    include Silo::Store
    silo :collection => "artist_attendance", :id => lambda { @artist.id }

    def initialize(artist)
      @artist = artist
    end

    def total_num_users_seen_live
      @artist.total_num_users_seen_live
    end

    def top_users_seen_live
      @artist.top_users_seen_live
    end

  end
end
```

SILOS are the sort-of ORM that stores our presenters' data in Mongo.

There are two things to declare: the collection name – usually based on the class being silod, and the id of the document – usually the same thing as an ActiveRecord model id.

This tells Silo that we want to store this object in Mongo.

```

module Presenters::Artists
  class AttendancePresenter

    include Silo::Store
    silo :collection => "artist_attendance", :id => lambda { @artist.id }

    def initialize(artist)
      @artist = artist
    end

    def total_num_users_seen_live
      @artist.total_num_users_seen_live
    end

    silo_method :total_num_users_seen_live

    def top_users_seen_live_pairs
      @artist.top_users_seen_live.map {|u| [u.id, u.num_events]}
    end

    silo_method :top_users_seen_live_pairs
  end
end

```

We tell Silo that we want to cache the slow methods so we annotate them with `silo_method`. This redefines methods behind the scenes so that when the object is asked in future for `total_num_users_seen_live`, say, it will first look in the Mongo document for a “total” key, and only if that is missing pass on to the method implementation of “total_num_users_seen_live” defined in the class.

Note that we had to change the definition of `top_users_seen_live_pairs`. MongoDB accepts JSON documents, so you have to turn your important data into JSON somehow. In this case what’s important is the User id, and the number of events for each of the Users. There is another method (not shown) for translating back to ActiveRecord models for the view to use.

This is less an ORM than a very simple database-backed method memoization technique.

The fall-through means that we can be sure the website will work (slowly) even if we haven’t managed to insert the right documents in the Silo beforehand, although that is always our goal.

Caching is easy.

But cache expiration is hard.

Let's make it easier.

with **THE SILOVATOR**

What I've shown you is great, but caching some data is not the hard part. Making sure that the data is expired and where appropriate pre-generated is where it gets tricky.

Website

```
class ArtistObserver
  observe Artist

  def after_create(artist)
    Silo.generate(presenter_for(artist))
  end

  def after_destroy(artist)
    Silo.remove(presenter_for(artist))
  end
end
```

Standard Rails observer. In this example implementation, we are conveniently forgetting that updates can happen. But the creation and destruction examples are real code.

When we tell `Silo` to generate and pass it an object implementing `Silo::Store`, the silos will internally reflect on the methods in the class that have `silos_method` annotations, and call the original implementations of those methods and then store the result in Mongo. Generally on object creations you end up with documents with a lot of zeros in them, and in this case that would be an empty list of users and a big 0 for number of users seen live.

Removing documents from the silo is fairly obvious.

This is nice, but doesn't scale. We may have to update 100s of silo in one of these, we can't afford to do them in process before the user sees a result.

In our system we replace this with...

Website

```
AsynchronousObservers.listen(Artist)
```

AMQP

CrUD

create
Artist
253846

CrUD

update
User
764

CrUD

destroy
Concert
546278

.... an out of process observer.

In the website all that is necessary is to tell the `AsynchronousObservers` what classes to listen to. It will hook into `after_*` callbacks for create, update and destroy operations and serialize the information about the CRUD operation into a message. It sends those messages over RabbitMQ to our silovator daemons.

This creates a ‘mega-feed’ of all the events that are happening in our system. This unifies **all** our background job processing into a single abstraction. If we need to create a new type of background job, we usually don’t have to even touch the front-end. We can spin up a new daemon that listens to any type of event.

For instance, we have email daemons that listen for `:create User` operations so we can send them a “Welcome to Songkick” email. We have image resizer daemons that listen for `:create Image` events.

Website

```
AsynchronousObservers.listen(Artist)
```

Silovators

```
class AttendanceSilovator < AsyncObserver

  listen :create, Artist do
    Silo.generate(presenter_for(artist))
  end

  listen :destroy, Artist do
    Silo.remove(presenter_for(artist))
  end
end
```

In the daemons we use this DSL for listening for CRUD operations and dispatching based on them.

This is the silovator code for regenerating the AttendancePresenter. It's similar to a Rails Observer but you can see there is some magic, e.g. the artist is available in the block. Also if it is an update operation you will have the ActiveRecord changes hash available for inspection. Again, this is real code.

We have a fleet of silovators running in production whose job it is to listen for all the events that could possibly change the denormalized data, and to update the Silos based on the changes. It takes a little bit of time for the changes to run across the message bus and through the silos, but that's measured in seconds at the moment.

Some of the silovator code is pretty complicated, because it doesn't save you from having to figure out what changes what. But adding a new denormalization is as simple as:

1. Adding silo_methods everywhere you want them.
2. Figuring out what can change the values of the data you just silod.
3. Writing a Silovator listener to update the silos based on that.

AttendanceSilovator Test

The silos should be updated correctly when

1. an Artist is created
2. an Artist is destroyed

Testing a silovator is important.

In our simplified example there are just two events that can change the silod data. So we could write out our test like this.

This is sort of the conceptual ‘plan’ of our test. To really test it we would have to go and write code to check that the right things happened.

But silos make it easy to translate this test into code.

What we really have are two things:

1. A set of data values attached to classes – the silos
2. A set of ‘events’ that can change that data.

Both of these can be declared and helpers can make our tests simpler.

```
describe AttendanceSilovator do
  before do
    silos { Artist.all.map {|a| AttendancePresenter.new(a)} }
  end

  silos_should_be_updated_when("an artist is created") do
    Factory(:artist)
  end

  describe "when an artist exists" do
    before do
      @artist = Factory(:artist)
    end

    silos_should_be_updated_when("an artist is destroyed") do
      @artist.destroy
    end
  end
end
```

This is a real test we have. And this is a complete test of the silovator we already discussed.

At the top the silos are declared.

And then we declare the events that can change them as blocks.

```
describe AttendanceSilovator do
  before do
    silos { Artist.all.map {|a| AttendancePresenter.new(a)} }
  end

  silos_should_be_updated_when("an artist is created") do
    Factory(:artist)
  end

  describe "when an artist exists" do
    before do
      @artist = Factory(:artist)
    end

    silos_should_be_updated_when("an artist is destroyed") do
      @artist.destroy
    end
  end
end
```

Smart helpers

The smart helpers generate rspec tests for the silovator.

And that's it. Behind the scenes these tests do a lot of work. But what it boils down to is this:

IF you have a correct and complete list of data-changing events
AND IF these tests pass,
THEN you can be sure silos are being expired correctly.

This saves us from a lot of boilerplate code that isn't really relevant to the essence of the test.

Questions now