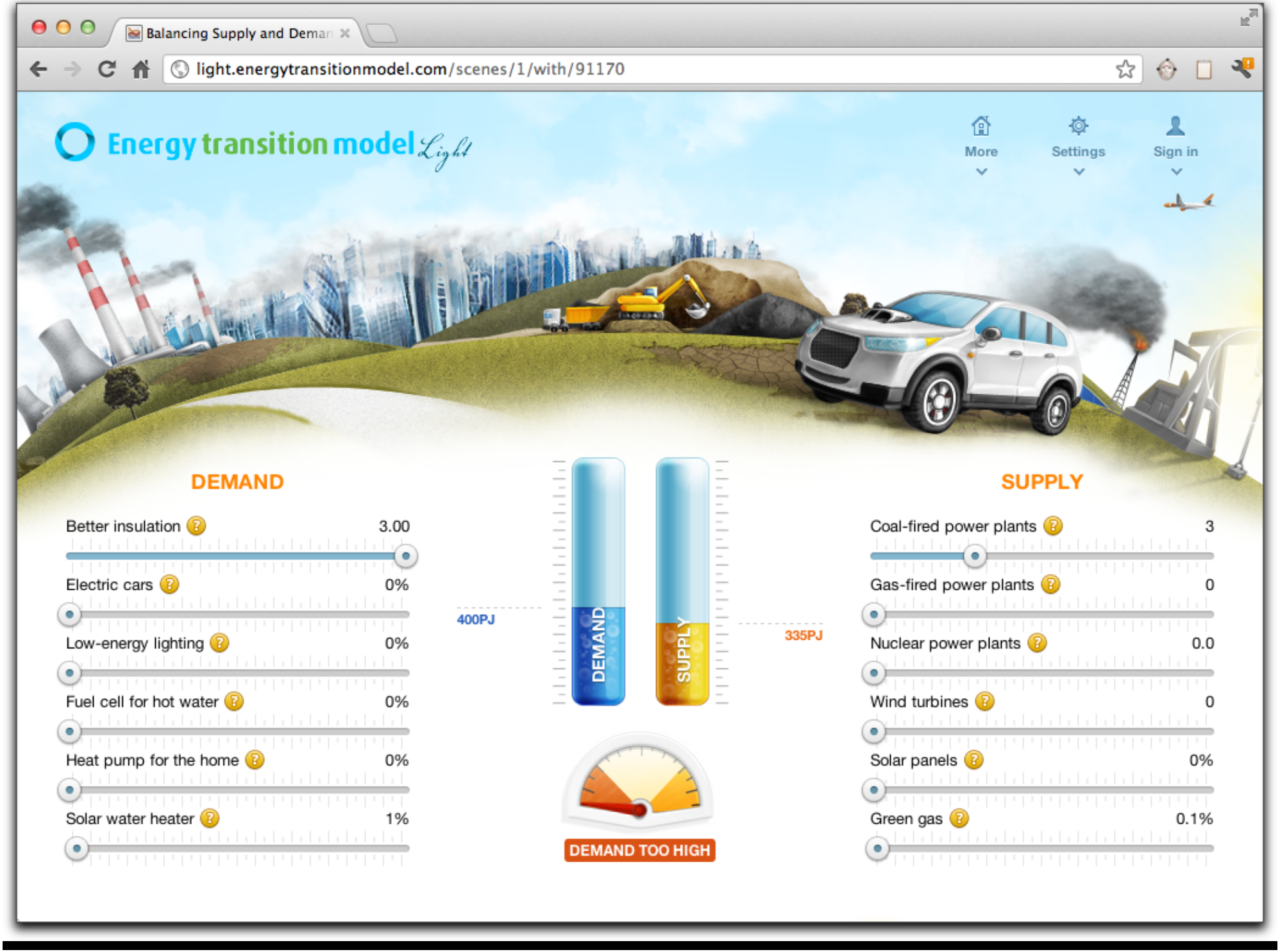
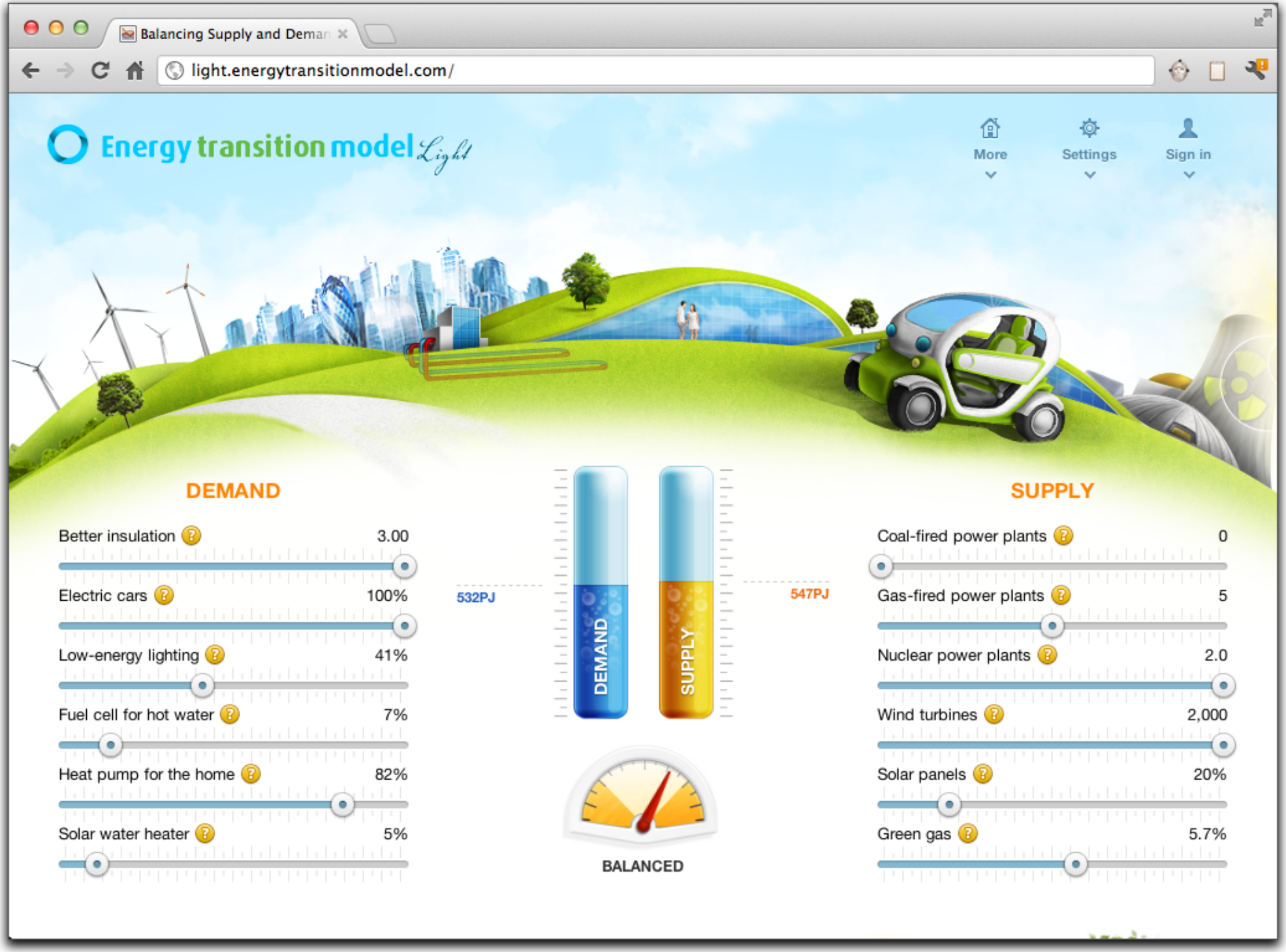
Run Ruby Run

Sebastian Burkhard
hasclass.com
twitter.com/hasclass?
github.com/hasclass?


Balancing Supply and Deman
light.energytransitionmodel.com/scenes/1/with/91170
Energy transition model Light
More
Settings
Sign in
DEMAND
Better insulation 3.00
Electric cars 0%
Low-energy lighting 0%
Fuel cell for hot water 0%
Heat pump for the home 0%
Solar water heater 1%
400PJ
DEMAND
SUPPLY
335PJ
DEMAND TOO HIGH
SUPPLY
Coal-fired power plants 3
Gas-fired power plants 0
Nuclear power plants 0.0
Wind turbines 0
Solar panels 0%
Green gas 0.1%


Balancing Supply and Deman
light.energytransitionmodel.com/
Energy transition model Light
More
Settings
Sign in
DEMAND
Better insulation 3.00
Electric cars 100%
Low-energy lighting 41%
Fuel cell for hot water 7%
Heat pump for the home 82%
Solar water heater 5%
532PJ
DEMAND
SUPPLY
547PJ
BALANCED
SUPPLY
Coal-fired power plants 0
Gas-fired power plants 5
Nuclear power plants 2.0
Wind turbines 2,000
Solar panels 20%
Green gas 5.7%

Summarized

```
"Fast ruby code is code that rarely runs.”
# Part 1: loading data
[ :cache,
  :cache_faster,
  :cache_more    ]
# Part 2: write faster ruby
def micro_optimize
def ugly_code
def stop_bad_habits
```

BlogController

```
def recommend_post
  r = Recommender.new(session[:user_history])
  r.load_posts_with_links          #~> 800ms
  r.load_meta_data                 #~> 500ms
  r.calculate                      #~> 200ms
  r.recommend_post                 #~>   5ms
end

before_filter :recommend_post
```

Trivial benchmark #1

```
class Post < ActiveRecord::Base
end

 10_000.times { Post.new        }
100_000.times { @post.rank = 1 }
```

Trivial benchmark #1

```
class Post < ActiveRecord::Base
end

 10_000.times { Post.new       }
100_000.times { @post.rank = 1 }

# 0.310  0.026  0.336 (0.421s)
# 0.750  0.010  0.760 (0.793s)
```

W.W.J.D.

What Would ~~Jesus~~ **Java** Do

PORO #1

Plain Old Ruby Object

- Fancy name for a plain vanilla ruby class
- Own namespace (e.g. Poro::Post)
- PORO class for every model class
- Every AR model gets a #to_poro
- All functionality is in POROs

PORO #1

```
def to_poro
  t = Poro::Post.new(
    id:   id,
    body: body
  )
end
```

```
module Poro
  class Graph
  class Post
  class Link
  class Tag
```

PORO #1

```
 10_000.times { Poro::Post.new }
100_000.times { post.rank = 1  }

#   0.000    0.010    0.010 (0.0035)
#   0.020    0.000    0.020 (0.0187)
```

loading records

```
r = Recommender.new(session[:user_history])
r.load_posts_with_links          #~> 800ms
r.load_meta_data                 #~> 500ms
r.calculate                      #~> 200ms
r.recommend_post                 #~>  10ms
```

```ruby
def load_posts_with_links
  posts = Post.all.to_poro.group_by(&:id)
  Link.all.to_poro.each do |link|
    l.source = posts[l.source_id]
    l.target = posts[l.target_id]
  end
  posts.values
end
```

dynamic_poros

```ruby
def load_posts_with_links
  Rails.cache.fetch("posts") do
    # ...
  end
end #~> 100ms

Rails.cache.fetch("posts#{Date.today}”)
Rails.cache.fetch(["posts", Date.today])
```

backstage

```
serialized = Marshal.dump( yield )
Rails.cache.write('posts', serialized)

Marshal.load(Rails.cache.read('posts'))


p1 = Marshal.load(Marshal.dump( posts ))
p2 = Marshal.load(Marshal.dump( posts ))
p1 != p2
```

cache with lazy-loading

```
def load_posts_with_links
  Rails.cache.fetch("posts") do
    Post.all.to_a
  end
end
```

```
r = Recommender.new(session[:user_history])
r.load_posts_with_links          #~> 100ms
r.load_meta_data                 #~> 500ms
r.calculate                      #~> 200ms
r.recomment_post                 #~>  10ms
```

caching meta_data

```
def load_meta_data
  {
    tags: Tag.all.to_poro,
    commenters: Commenter.all.to_poro
  }
end
```

NastyCache

```
NastyCache.fetch(”meta") { load_meta_data }
#~>    0ms
```

NastyCache

```
NastyCache.fetch(”meta") { load_meta_data }
#~>   0ms


class NastyCache
  @cache_store = {}

  def self.fetch(key)
    @cache_store[key] ||= yield
  end
end
```

NastyCache Logs

```
server_1 (0.500s): NC#fetch: meta
server_1 (0.000s): NC#fetch: meta
server_2 (0.500s): NC#fetch: meta
server_2 (0.000s): NC#fetch: meta
server_3 (0.500s): NC#fetch: meta
server_3 (0.000s): NC#fetch: meta
server_3 (0.000s): NC#fetch: meta
server_2 (0.000s): NC#fetch: meta
```

combine NastyCache & Rails.cache #3

```
def self.fetch_cached(key)
  @cache[key] ||= Rails.cache.fetch(key) do
    yield
  end
end
```

```
server_1 (0.500s): NS#fetch_cached: meta
server_1 (0.000s): NS#fetch_cached: meta
server_2 (0.100s): NS#fetch_cached: meta
server_2 (0.000s): NS#fetch_cached: meta
server_3 (0.100s): NS#fetch_cached: meta
```

Rails.cache vs NastyCache #3

```
cache_1 = Rails.cache.fetch("meta") { }
cache_2 = Rails.cache.fetch(”meta") { }
cache_1 != cache_2

nasty_1 = NastyCache.fetch("meta") { }
nasty_2 = NastyCache.fetch("meta") { }
nasty_1 == nasty_2
```

NastyCache #3

```
posts = NastyCache.fetch(”posts") { ... }
posts.first.rank # => 0

# server(1). request 1
posts.first.rank = 3

# server(1). request 2
posts.first.rank #=> 3

# server(2). request 1
posts.first.rank #=> 0
```

cache_more

```
r = Recommender.new(session[:user_history])
r.load_posts_with_links          #~> 100ms
r.load_meta_data                 #~> 001ms
r.calculate                      #~> 200ms
r.recomment_post                 #~>  10ms
```

back to the stateful part

```
class Poro::Post
  attr_accessor :graph, :body, :links, :rank

  def calculate
    rank  = links.map(&:juice).sum
    rank += 10 if visited?
  end
```

extract state #4

```
class Poro::Post
  attr_accessor :graph, :body, :links, :rank

  def calculate
    rank  = links.map(&:juice).sum
    rank += 10 if visited?
  end
```

get/set into dataset #4

```
@recommender.state = { }
#=> {PORO_ID: {rank: 3, boost: 1.2}, ...}
def rank
  @recommender.state[id][:rank]
end

def rank=(v)
  @recommender.state[id][:rank] = v
end

dataset_accessor :rank    # generalized
```

nasty_cache object_graph #4

```
def load_posts_with_links
  NastyCache.fetch_cached("posts") {...}
end


r = Recommender.new(session[:user_history])
r.load_posts_with_links          #~> 005ms
r.load_meta_data                 #~> 001ms
r.state = {}
r.calculate                      #~> 250ms
```

```
# check if still threadsafe...
# config.threadsafe!
```

```
Completed OK 0.401ms
Completed OK 0.403ms
Completed OK 1.000ms
Completed OK 0.405ms
Completed OK 0.399ms
Completed OK 1.100ms
Completed OK 0.405ms
...
```

Quickfix GC#disable

```
class ApiController
  around_filter :disable_gc

  def disable_gc
    GC::disable
    yield
  ensure
    GC::enable
  end

  def show
    @recommended.to_json
  end
```

WRITE EFFICIENT RUBY

avoid small classes

```
class Poro::Tag
  attr_reader :name
end

class Poro::Category
  attr_reader :name
end
# work with the name as symbols
```

memoize

```ruby
def mem_simple
  @mem_simple ||= 1
end # 1_000_000 : 0.25s

def mem_falsy
  unless defined?(@mem_falsy)
    @mem_falsy = 1
  end
  @mem_falsy
end # 1_000_000 : 0.35s
```

memoize

```
# something in between
def mem_hash
  unless @memo.has_key?(:mem_hash)
    @memo[:mem_hash] = 1
  end
  @memo[:mem_hash]
end  # 0.29s
```

cache me if you can

```
class Poro::Post
  def initialize
    cache_me
  end

  def word_count
    @word_count ||= @body.split(" ").length
  end

  def cache_me
    word_count
  end
```

Don’t be lazy

```
hsh = {}

0.upto(1_000) do |x|
  hsh[x] ||= {}

  0.upto(1_000) do |y|
    hsh[x][y] = 1
  end
end
```

```ruby
hsh = {}

0.upto(1_000) do |x|
  hsh[x] ||= {}
  row = hsh[x]
  0.upto(1_000) do |y|
    row[y] = 1
  end
end
```

understand your methods

```
(0..1000).select(&:odd?).first
# make new array with only odd numbers.
# then return the first number.

(0..1000).detect(&:odd?)
# return first odd number
```

```ruby
arr.each do |obj|
  arr.delete(obj) if obj.odd?
end


# delete_at 10x faster
arr.each_with_index do |obj, i|
  arr.delete_at(i) if obj.odd?
end
```

avoid method_missing chains

```
def method_missing(name, *args)
  name = name.to_s
  if m = name.match(/tagged_with_(.*)/)
  elsif m = name.match(/find_(.*)/)
  elsif m = name.match(/search_(.*)/)
  ...
end
```

cannot?

```
def method_missing(name, *args)
  if m = name.to_s.match(/tagged_with_(.*)/)
    tag = m.last
    posts.select{|p| p.tags.include?(tag)}
  elsif …
end
```

can! meta^{2}programming

```
def method_missing(name, *args)
  if m = name.to_s.match(/tagged_with_(.*)/)
    tag = m.last
    self.class.send(:define_method, tag) do
      posts.select{|p| p.tags.include?(tag)}
    end
  end
end
```

Revised on next slide

can! meta²programming

```
REVISED. Thanks to @gnufied
# a) Faster and not messing up scope

def method_missing(name, *args)
  if m = name.to_s.match(/tagged_with_(.*)/)
    tag = m.last
    self.class_eval <<-EOF,__FILE__,__LINE__ +1
      def #{tag}
        posts.select {|p| p.tags.include?('#{tag}') }
      end
    EOF
  end
end
```

m_m makes flatten slow #5

```ruby
class Foo
  def method_missing(name, *args)
    # ...
  end
end

foos = (0..10_000).map{ Foo.new }
foos.flatten # => wait
```

method_missing more

```
class Foo
  def to_ary; nil end

  def method_missing(name, *args)
    # ...
  end
end

foos = (0..10_000).map{ Foo.new }
foos.flatten # => quick
```

method_missing more

```
class Foo
  attr_reader :to_ary

  def method_missing(name, *args)
    # ...
  end
end

foos = (0..10_000).map{ Foo.new }
foos.flatten # => quick
```

beware inspect of death

```
class Link
  attr_accessor :source, :target

class Post
  attr_accessor :inlinks, :out

object_graph.inspect
#=> death by circular reference
```

beware inspect of death

```
class Poro::Link
  def inspect() "<Link ...>" end

class Poro::Post
  def inspect() "<Post ...>" end
```

where to go from here

```
public class PostPojo {
  public int word_count;

  public PostPojo() { ... }
  public int calculate_rank() { ... }
  // everything that needs to run fast
}

class Poro::Post < PostPojo
  def things_that_can_be_slow; end
end
```