

Rails' Next Top Model

Adam Keys, expert typist at Gowalla

<http://therealadam.com>

@therealadam

RailsConf 2010

Hi, I'm Adam Keys. I'm an expert typist at Gowalla and an amateur language lawyer. Today I'm going to talk about what I consider the most intriguing part of the reimagination of Rails that is Rails 3. Namely, I want to explore how ActiveRecord was extracted from itself into ActiveModel and ActiveRelation.



Four Great Looks!



- * Extractions reduce friction in building little languages on top of data stores
- * Reduce the boilerplate code involved in bringing up a data layer
- * Make it easier to add some of the things we've come to take for granted
- * Allow developers to focus on building better APIs for data

Clean up your domain objects

ActiveSupport fanciness



- * ActiveSupport, the oft-maligned cake on top of ActiveRecord and Rails are built
- * Smaller and less cumbersome in Rails 3, cherry-pick the functionality you want
- * Use ActiveSupport instead of rolling your own extensions or copy-paste reuse
- * Tighten up your classes by extracting concerns

```
require 'common'
require 'active_support/inflector'
require 'active_support/cache'

class User

  attr_accessor :name

  def friends
    cache.fetch("user-#{name}-friends") do
      %w{ Peter Egon Winston }
    end
  end

  protected

  def cache
    ActiveSupport::Cache::MemCacheStore.new
  end

end
```

- * Not too different from the user model in your own applications
- * `cache` is the simplest thing that might work, but could we make it better and cleaner?


```
require 'active_support/core_ext/class'

class User
  cattr_accessor :cache
  attr_accessor :name

  def friends
    cache.fetch("user-#{name}-friends") do
      %w{ Peter Egon Winston }
    end
  end
end

end
```

- * Use a class attribute to get the cache configuration out of the instance
- * Could use the inheritable version if we are building our own framework

```
User.cache = ActiveSupport::Cache::MemCacheStore.new
```

* In our application setup, create a cache instance and assign it to whatever classes need it

```
def friends
  cache.fetch("user-#{name}-friends") do
    %w{ Peter Egon Winston }
  end
end
```

- * Suppose we're going to end up with a lot of methods that look like this
- * There's a lot of potential boiler-plate code to write there
- * Is there a way we can isolate specify a name, key format, and the logic to use?

```
cache_key(:friends, :friends_key) do
  %w{ Peter Egon Winston }
end

def friends_key
  "user-#{name}-friends"
end
```

- * I like to start by thinking what the little language will look like
- * From there, I start adding the code to make it go
- * Hat tip, Rich Kilmer


```
cattr_accessor :cache_lookups, :cache_keys do
  {}
end
```

```
def self.cache_key(name, key, &block)
  class_eval %Q{
    cache_lookups[name] = block
    cache_keys[name] = key

    def #{name}
      return @#{name} if @#{name}.present?
      key = method(cache_keys[:#{name}]).call
      @#{name} = cache.fetch(key) do
        block.call
      end
    end
  }
end
```

- * Add a couple class attributes to keep track of things, this time with default values
- * Write a class method that adds a method for each cache key we add
- * Look up the the cache key to fetch from, look up the body to call to populate it, off we go
- * The catch: block is bound to class rather than instance

```

class User
  cattr_accessor :cache
  attr_accessor :name

  cattr_accessor :cache_lookups, :cache_keys do
    {}
  end

  def self.cache_key(name, key, &block)
    class_eval %Q{
      cache_lookups[name] = block
      cache_keys[name] = key

      def #{name}
        return @#{name} if @#{name}.present?
        key = method(cache_keys[:#{name}]).call
        @#{name} = cache.fetch(key) do
          block.call
        end
      end
    }
  end

  cache_key(:friends, :friends_key) do
    %w{ Peter Egon Winston }
  end

  def friends_key
    "user-#{name}-friends"
  end
end

```

- * Downside: now our class won't fit nicely on one slide; is this a smell?
- * ActiveSupport enables a nice little refactoring I've started calling "extract concern"

```

class User
  cattr_accessor :cache
  attr_accessor :name

  cattr_accessor :cache_lookups, :cache_lookups_key
  {}
end

def self.cache_lookups_key
  class_name = self.name
  cache_key = "tyra-#{name}-friends"
end
end

```



- * Downside: now our class won't fit nicely on one slide; is this a smell?
- * ActiveSupport enables a nice little refactoring I've started calling "extract concern"

```

require 'active_support/concern'

module Cacheably
  extend ActiveSupport::Concern

  included do
    attr_accessor :cache

    attr_accessor :cache_lookups, :cache_keys do
      {}
    end

    def self.cache_key(name, key, &block)
      cache_lookups[name] = block
      cache_keys[name] = key

      class_eval %Q{
        def #{name}
          return @#{name} if @#{name}.present?
          key = method(cache_keys[:#{name}]).call
          @#{name} = cache.fetch(key) do
            block.call
          end
        end
      }
    end
  end
end
end

```

- * We pick up all the machinery involved in making `cache_key` work and move into a module
- * Then we wrap that bit in the included hook and extend ActiveSupport::Concern
- * Easier to read than the old convention of modules included in, ala plugins

```
class User
  include Cacheably

  attr_accessor :name

  cache_key(:friends, :friends_key) do
    %w{ Peter Egon Winston }
  end

  def friends_key
    "user-#{name}-friends"
  end
end
```

- * Domain object fits on one slide again
- * Easy to see where the cache behavior comes from

Accessors + concerns = slimming effect

- * ActiveSupport can help remove tedious code from your logic
- * ActiveSupport can make your classes simpler to reason about
- * Also look out for handy helper classes like MessageVerifier/Encryper, SecureRandom, etc.
- * Give it a fresh look, even if it's previously stabbed you in the face

Models that look good and want to talk good too

ActiveModel validations



- * ActiveModel is the result of extracting much of the goodness of ActiveRecord
- * If you've ever wanted validations, callbacks, dirty tracking, or serialization, this is your jam
- * Better still, ActiveModel is cherry-pickable like ActiveSupport


```
include ActiveRecord::Validations
```

```
validates_presence_of :name
```

```
validates_length_of :name,
```

```
  :minimum => 3,
```

```
  :message => 'Names with less than 3 characters are dumb'
```

- * Adding validations to our user model is easy
- * These are one in the same with what you're using in AR
- * No methods needed to get this functionality; just include and you're on your way

```
class GhostbusterValidator < ActiveRecord::Validator

  def validate(record)
    names = %w{ Peter Ray Egon Winston }
    return if names.include?(record.name)
    record.errors[:base] << "Not a Ghostbuster :("
  end

end
```

- * With ActiveRecord, we can also implement validation logic in external classes
- * Nice for sharing between projects or extracting involved validation

`validates_with GhostbusterValidator`

- * Step 1: specify your validation class
- * There is no step 2

```
class User
  include Cacheably

  attr_accessor :name

  cache_key(:friends, :friends_key) do
    %w{ Peter Egon Winston }
  end

  def friends_key
    "user-#{name}-friends"
  end

  include ActiveRecord::Validations

  validates_presence_of :name
  validates_length_of :name,
    :minimum => 3,
    :message => 'Names with less than 3
characters are dumb'
  validates_with GhostbusterValidator

end
```

- * Now our class looks like this
- * Still fits on one slide

```
class User
  include Cacheably

  attr_accessor :name

  cache_key(:friends, :friends_key) do
    %w{ Peter Egon Winston }
  end

  def friends_key
    "user-#{name}-friends"
  end

  include ActiveRecord::Validations

  validates_presence_of :name
  validates_length_of :name,
    :minimum => 3,
    :message => 'Names with less than 3
characters are dumb'
  validates_with GhostbusterValidator

end
```



- * Now our class looks like this
- * Still fits on one slide

```
>> u = User.new
=> #<User:0x103a56f28>
>> u.valid?
=> false
>> u.errors
=> #<OrderedHash {:base=>["Not a Ghostbuster :("], :name=>["can't be blank", "can't be blank", "Names with less than 3 characters are dumb", "can't be blank", "Names with less than 3 characters are dumb"]}>
```

Using the validations, no surprise, looks just like AR

```
>> u.name = 'Ron'
=> "Ron"
>> u.valid?
=> false
>> u.errors
=> #<OrderedHash {:base=>["Not a Ghostbuster :("]}>
```

Ron Evans is a cool dude, but he's no Ghostbuster


```
>> u.name = 'Ray'  
=> "Ray"  
>> u.valid?  
=> true
```

Ray is a bonafide Ghostbuster

Serialize your objects, your way

ActiveModel lifecycle helpers



- * Validations are cool and easy
- * What if we want to encode our object as JSON or XML
- * Tyra is not so sure she wants to write that code herself

```
attr_accessor :degree, :thought
```

Let's add a couple more attributes to our class, for grins.

```
def attributes  
  @attributes ||= { 'name' => name,  
                    'degree' => degree,  
                    'thought' => thought }  
end
```

```
def attributes=(hash)  
  self.name = hash['name']  
  self.degree = hash['degree']  
  self.thought = hash['thought']  
end
```

- * If we add `attributes`, AMo knows what attributes to serialize when it encodes your object
- * If we implement `attributes=`, we can specify how a serialized object gets decoded

```
include ActiveRecord::Serializers::JSON  
include ActiveRecord::Serializers::Xml
```

Once that's done, we pull in the serializers we want to make available.

```
>> u.serializable_hash
=> {"name"=>"Ray Stanz", "degree"=>"Parapsychology", "thought"=>"The
Stay-Puft Marshmallow Man"}
>> u.to_json
=> "{\"name\":\"Ray Stanz\",\"degree\":\"Parapsychology\",\"thought
\":\"The Stay-Puft Marshmallow Man\"}"
>> u.to_xml
=> "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<user>\n
<degree>Parapsychology</degree>\n  <name>Ray Stanz</name>\n
<thought>The Stay-Puft Marshmallow Man</thought>\n</user>\n"
```

- * We get a serializable hash method which is what gets encoded
- * `to_json` and `to_xml` are now ours, just like with AR

```
>> json = u.to_json
=> '{"name":"Ray Stanz","degree":"Parapsychology","thought":"The Stay-Puft Marshmallow Man"}'
>> new_user = User.new
=> #<User:0x103166378>
>> new_user.from_json(json)
=> #<User:0x103166378 @name="Ray Stanz", @degree="Parapsychology", @thought="The Stay-Puft Marshmallow Man">
```

We can even use `from_json` and `from_xml`!

Persistence and queries, like a boss

ActiveRelation persistence



- * In Rails 2, AR contains a bunch of logic for banging string together to form queries
- * In Rails 3, that's been abstracted into a library that models the relational algebra that databases use
- * But, Arel makes it possible to use the same API to query all your data sources

```
include Arel::Relation  
  
cattr_accessor :engine
```

- * To make our class query and persist like an AR object, we need to include `Relation`
- * We'll also need an engine, which we'll look into soon
- * Including Relation gives us a whole bunch of methods that look familiar from AR: where, order, take, skip, and some that are more SQLish: insert, update, delete

```
def save
  insert(self)
end

def find(name)
  key = name.downcase.gsub(' ', '_')
  where("user-#{key}").call
end
```

- * Since our goal is to be somewhat like AR, we'll add some sugar on top of the Relation
- * Our save isn't as clever as AR's, in that it only creates records, but we could add dirty tracking later
- * Find is just sugar on top of `where`, which is quite similar to how we use it in AR

```
def marshal_dump  
  attributes  
end
```

```
def marshal_load(hash)  
  self.attributes = hash  
end
```

- * For those following along at home, we'll need these on User too, due to some oddness between AMo's serialization and Arel's each method
- * Ideally, we'd serialize with JSON instead, but this gets the job done for now

```
class UserEngine  
  
  attr_reader :cache  
  
  def initialize(cache)  
    @cache = cache  
  end  
  
end
```

- * Arel implements the query mechanism, but you still need to write an “engine” to handle translating to the right query language and reading/writing
- * These seem to be called engines by convention, but they are basically just a duck type
- * The methods we’ll need to implement are our good CRUD friends

```
def create(insert)
  record = insert.relation
  key = record.cache_key
  value = record
  cache.write(key, value)
end
```

- * Getting these engines up is mostly a matter of grokking what is in an ARel relation
- * Everything is passed a relation
- * The insert object has a relation that represents the record we want to create
- * Uses Marshal, JSON or YAML would be nicer

```
def read(select)
  raise ArgumentError.new("#{select.class} not
    supported") unless select.is_a?(Arel::Where)
  key = select.predicates.first.value
  cache.read(key)
end
```

- * Reads are where we query the datastore
- * The select object contains the last method in whatever query chain we called
- * Our memcached-based gizmo only supports `where` but we could get a `project`, `take`, `order` etc.
- * Spend lots of time poking the insides of these various objects to grab the data you need to construct a query


```
def update(update)
  record = update.assignments.value
  key = record.cache_key
  value = record
  cache.write(key, value)
end
```

- * Update objects contain an assignment, which has the record we're after
- * Again, uses Marshal, which is suboptimal

```
def delete(delete)
    key = delete.relation.cache_key
    cache.delete(key)
end
```

* Delete passes the relation we're going to remove; not much going on here

```

class UserEngine

  attr_reader :cache

  def initialize(cache)
    @cache = cache
  end

  def create(insert)
    record = insert.relation
    key = record.cache_key
    value = record
    cache.write(key, value)
  end

  def read(select)
    raise ArgumentError.new("#{select.class} not supported") unless select.is_a?(Arel::Where)
    key = select.predicates.first.value
    cache.read(key)
  end

  def update(update)
    record = relation.assignments.value
    key = record.cache_key
    value = record
    cache.write(key, value)
  end

  def delete(delete)
    key = delete.relation.cache_key
    cache.delete(key)
  end

end

```

- * The entirety of our engine
- * Use this as a starting point for your datastore; it's probably not entirely right for what you want to do, but it's better than trying to figure things out from scratch
- * Read the in-memory engine that comes with ARel or look at arel-mongo

```
User.cache = ActiveSupport::Cache::MemCacheStore.new  
User.engine = UserEngine.new(User.cache)
```

Here's how we set up our engine

```
# >> u = User.new
# => #<User:0x103655eb0>
# >> u.name = 'Ray Stanz'
# => "Ray Stanz"
# >> u.degree = 'Parapsychology'
# => "Parapsychology"
# >> u.thought = 'The Stay-Puft Marshmallow Man'
# => "The Stay-Puft Marshmallow Man"
# >> u.save
# => true
# >> other = User.new
# => #<User:0x103643b20>
# >> user.find('Ray Stanz')
# => #<User:0x10363f958 @name="Ray Stanz", @degree="Parapsychology",
@thought="The Stay-Puft Marshmallow Man">
# >> user.thought = ''
# => ""
# >> user.update(user)
# => true
# >> user.delete
# => true
```

- * Create a user object
- * Save it to the cache
- * Read it back out
- * Update it
- * Delete it

```
# >> u = User.new
# => #<User:0x103655eb0>
# >> u.name = 'Ray Stanz'
# => "Ray Stanz"
# >> u.degree = 'Parapsychology'
# => "Parapsychology"
# >> u.thought = 'The Stay-Puft Marshmallow Man'
# => "The Stay-Puft Marshmallow Man"
# >> u.save
# => true
# >> other = User.new
# => #<User:0x103643b20>
# >> user.find('Ray Stanz')
# => #<User:0x10363f958 @name="Ray Stanz", @degree="Parapsychology",
@thought="The Stay-Puft Marshmallow Man">
# >> user.thought = ''
# => ""
# >> user.update(user)
# => true
# >> user.delete
# => true
```



- * Create a user object
- * Save it to the cache
- * Read it back out
- * Update it
- * Delete it

```
# >> u = User.new
# => #<User:0x103655eb0>
# >> u.name = 'Ray Stanz'
# => "Ray Stanz"
# >> u.degree = 'Parapsychology'
# => "Parapsychology"
# >> u.thought = 'The Stay-Puft Marshmallow Man'
# => "The Stay-Puft Marshmallow Man"
# >> u.save
# => true
# >> other = User.new
# => #<User:0x103643b20>
# >> user.find('Ray Stanz')
# => #<User:0x10363f958 @name="Ray Stanz", @degree="Parapsychology",
@thought="The Stay-Puft Marshmallow Man">
# >> user.thought = ''
# => ""
# >> user.update(user)
# => true
# >> user.delete
# => true
```

- * Create a user object
- * Save it to the cache
- * Read it back out
- * Update it
- * Delete it



~15 collars, BTW

- * One dude, one pair of shades, one fleshbeard, fifteen collars
- * We've popped a lot of collars, but we got a lot of functionality too


```

class User
  include Cacheably

  attr_accessor :name

  cache_key(:friends, :friends_key) do
    %w{ Peter Egon Winston }
  end

  def friends_key
    "user-#{name}-friends"
  end

  include ActiveRecord::Validations

  validates_presence_of :name
  validates_length_of :name, :minimum => 3, :message => 'Names with less than 3 characters are dumb'
  validates_with GhostbusterValidator

  include ActiveRecord::Serialization

  attr_accessor :degree, :thought

  def attributes
    @attributes ||= {'name' => name, 'degree' => degree, 'thought' => thought}
  end

  def attributes=(hash)
    self.name = hash['name']
    self.degree = hash['degree']
    self.thought = hash['thought']
  end

  include ActiveRecord::Serializers::JSON
  include ActiveRecord::Serializers::Xml

  include Arel::Relation

  cattr_accessor :engine

  # Our engine uses this method to infer the record's key
  def cache_key
    "user-#{name.downcase.gsub(' ', '_')}"
  end

  def marshal_dump
    attributes
  end

  def marshal_load(hash)
    self.attributes = hash
  end

  def save
    # HAX: use dirty tracking to call insert or update here
    insert(self)
  end

  def find(name)
    key = name.downcase.gsub(' ', '_')
    where("user-#{key}").call
  end
end

```

- * Here’s our domain model and here’s all the support code
- * What we get: declarative lazy caching, validations, serialization, persistence, querying

```

class User
  include Cacheabilly

  attr_accessor :name

  cache_key(:friends, :friends_key) do
    %w{ Peter Egon Winston }
  end

  def friends_key
    "user-#{name}-friends"
  end

  include ActiveModel::Validations

  validates_presence_of :name
  validates_length_of :name, :minimum => 3, :message => 'Names with less than 3 characters are dumb'
  validates_with GhostbusterValidator

  include ActiveModel::Serialization

  attr_accessor :degree, :thought

  def attributes
    @attributes ||= {'name' => name, 'degree' => degree, 'thought' => thought}
  end

  def attributes=(hash)
    self.name = hash['name']
    self.degree = hash['degree']
    self.thought = hash['thought']
  end

  include ActiveModel::Serializers::JSON
  include ActiveModel::Serializers::Xml

  include Arel::Relation

  attr_accessor :engine

  # Our engine uses this method to infer the record's key
  def cache_key
    "user-#{name.downcase.gsub(' ', '_')}"
  end

  def marshal_dump
    attributes
  end

  def marshal_load(hash)
    self.attributes = hash
  end

  def save
    # HAX: use dirty tracking to call insert or update here
    insert(self)
  end

  def find(name)
    key = name.downcase.gsub(' ', '_')
    where("user-#{key}").call
  end

end

```

```

require 'common'
require 'active_support/concern'
require 'active_support/core_ext/class'
require 'active_support/inflector'
require 'active_support/cache'
require 'active_model'
require 'arel'

module Cacheabilly
  extend ActiveSupport::Concern

  included do
    attr_accessor :cache

    attr_accessor :cache_lookups, :cache_keys do
      {}
    end

    def self.cache_key(name, key, &block)
      cache_lookups[name] = block
      cache_keys[name] = key

      class_eval %Q{
        def #{name}
          return @#{name} if @#{name}.present?
          key = method(cache_keys[:#{name}]).call
          @#{name} = cache.fetch(key) do
            block.call
          end
        end
      }
    end
  end
end

class GhostbusterValidator < ActiveModel::Validator

  def validate(record)
    return if %w{ Peter Ray Egon Winston }.include?(record.name)
    record.errors[:base] << "Not a Ghostbuster :("
  end

end

```

- * Here’s our domain model and here’s all the support code
- * What we get: declarative lazy caching, validations, serialization, persistence, querying

```

class User
  include Cacheabilly

  attr_accessor :name

  cache_key(:friends, :friends_key) do
    %w{ Peter Egon Winston }
  end

  def friends_key
    "user-#{name}-friends"
  end

  include ActiveModel::Validations

  validates_presence_of :name
  validates_length_of :name, :minimum => 3, :message => 'Names with less than 3 characters are dumb'
  validates_with GhostbusterValidator

  include ActiveModel::Serialization

  attr_accessor :degree, :thought

  def attributes
    @attributes ||= {'name' => name, 'degree' => degree, 'thought' => thought}
  end

  def attributes=(hash)
    self.name = hash['name']
    self.degree = hash['degree']
    self.thought = hash['thought']
  end

  include ActiveModel::Serializers::JSON
  include ActiveModel::Serializers::Xml

  include Arel::Relation

  attr_reader :engine

  # Our engine uses this method to infer the record's key
  def cache_key
    "user-#{name.downcase.gsub(' ', '_')}"
  end

  def marshal_dump
    attributes
  end

  def marshal_load(hash)
    self.attributes = hash
  end

  def save
    # HAX: use dirty tracking to call insert or update here
    insert(self)
  end

  def find(name)
    key = name.downcase.gsub(' ', '_')
    where("user-#{key}").call
  end
end

```

```

require 'common'
require 'active_support/concern'
require 'active_support/core_ext/class'
require 'active_support/inflector'
require 'active_support/cache'
require 'active_model'
require 'arel'

module Cacheabilly
  extend ActiveSupport::Concern

  included do
    attr_accessor :cache

    attr_accessor :cache_lookups, :cache_keys do
      {}
    end

    def self.cache_key(name, key, &block)
      cache_lookups[name] = block
      cache_keys[name] = key

      class_eval %Q{
        def #{name}
          return @#{name} if @#{name}.present?
          key = method(cache_keys[:#{name}]).call
          @#{name} = cache.fetch(key) do
            block.call
          end
        end
      }
    end
  end
end

class GhostbusterValidator < ActiveModel::Validator

  def validate(record)
    return if %w{ Peter Ray Egon Winston }.include?(record.name)
    record.errors[:base] << "Not a Ghostbuster :("
  end
end

class UserEngine

  attr_reader :cache

  def initialize(cache)
    @cache = cache
  end

  def create(insert)
    record = insert.relation
    key = record.cache_key
    value = record # Note: this uses Marshal, b/c to_json w/ arel is buggy
    cache.write(key, value)
  end

  # Ignores chained queries, i.e. take(n).where(...)
  def read(select)
    raise ArgumentError.new("#{select.class} not supported") unless select.is_a?(Arel::Where)
    key = select.predicates.first.value
    cache.read(key)
  end

  def update(update)
    record = relation.assignments.value
    key = record.cache_key
    value = record
    cache.write(key, value)
  end

  def delete(delete)
    key = delete.relation.cache_key
    cache.delete(key)
  end
end

User.cache = ActiveSupport::Cache::MemCacheStore.new('localhost')
User.engine = UserEngine.new(User.cache)

```

- * Here’s our domain model and here’s all the support code
- * What we get: declarative lazy caching, validations, serialization, persistence, querying

“**Keys**” to success

- give ActiveSupport a try
- fancy up your classes with AMo
- build data layers with ARel
- make better codes



Thanks!

I hope you all got something out of this talk. I'll post the slides and example code on my website, therealadam.com, soon. Thanks for coming!