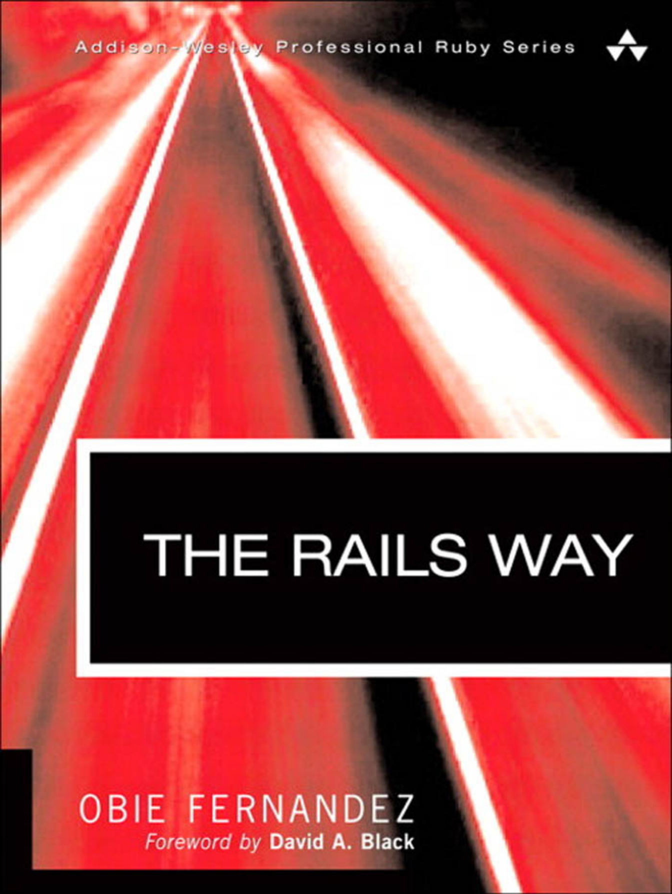

Addison-Wesley Professional Ruby Series

THE RAILS WAY

OBIE FERNANDEZ

Foreword by **David A. Black**

"I can positively say that it's the single best Rails book ever published to date. By a long shot."

—Antonio Cangiano, Software Engineer and Technical Evangelist at IBM

This book is a great crash course in Ruby on Rails! It doesn't just document the features of Rails, it filters everything through the lens of an experienced Rails developer—so you come out a Pro on the other side.

—Dirk Elmendorf, co-founder of Rackspace & Rails developer since 2005

"The key to The Rails Way is in the title. It literally covers the "way" to do almost everything with Rails. Writing a truly exhaustive reference to the most popular Web application framework used by thousands of developers is no mean feat. A thankful community of developers that has struggled to rely on scant documentation will embrace *The Rails Way* with open arms. A tour de force!"

—Peter Cooper, Editor, Ruby Inside

"In the past year, dozens of Rails books have been rushed to publication. A handful are good. Most regurgitate rudimentary information easily found on the Web. Only this book provides both the broad and deep technicalities of Rails. Nascent and expert developers, I recommend you follow *The Rails Way.*"

—Martin Streicher, Chief Technology Officer,
McClatchy Interactive, former Editor-in-Chief of Linux Magazine

"Hal Fulton's *The Ruby Way* has always been by my side as a reference while programming Ruby. Many times I had wished there was a book that had the same depth and attention to detail, only focused on the Rails framework. That book is now here and hasn't left my desk for the past month."

—Nate Klaiber, Ruby Programmer

"As noted in my contribution to the Afterword: "What Is the Rails Way (To You)?," I knew soon after becoming involved with Rails that I had found something great. Now, with Obie's book, I have been able to step into Ruby on Rails development coming from .NET and be productive right away. The applications I have created I believe to be a much better quality due to the techniques I learned using Obie's knowledge."

—Robert Bazinet, InfoQ.com, .NET and Ruby community Editor,
and founding member of the Hartford, CT Ruby Brigade

"Extremely well written; it's a resource that every Rails programmer should have. Yes, it's that good."

—Reuven Lerner, Linux Journal columnist

the rule that a timesheet approver must be authorized, you'd need to add a `before_save` callback to the `Timesheet` class itself. Callbacks are covered in detail at the beginning of Chapter 9, "Advanced `ActiveRecord`," and since I've gotten us a little bit off on a tangent, we'll go back to the list of options available for the `belongs_to` association.

:foreign_key

Specifies the name of the foreign key column that should be used to find the associated object. Rails will normally infer this setting from the name of the association, by adding `_id` . You can override the inferred foreign key name with this option if necessary.

```
# without the explicit option, Rails would guess administrator_id
belongs_to :administrator, :foreign_key => 'admin_user_id'
```

:counter_cache

Use this option to make Rails automatically update a counter field on the associated object with the number of belonging objects. The option value can be `true`, in which case the *pluralized* name of the belonging class plus `_count` is used, or you can supply your own column name to be used:

```
:counter_cache => true
:counter_cache => 'number_of_children'
```

If a significant percentage of your association collections will be empty at any given moment, you can optimize performance at the cost of some extra database storage by using counter caches liberally. The reason is that when the counter cache attribute is at zero, Rails *won't even try* to query the database for the associated records!

NOTE

The value of the counter cache column *must be set to zero by default* in the database! Otherwise the counter caching *won't work at all.* It's because the way that Rails implements the counter caching behavior is by adding a simple callback that goes directly to the database with an `UPDATE` command and increments the value of the counter.

If you're not careful, and neglect to set a default value of 0 for the counter cache column on the database, or misspell the column name, the counter cache will still seem to work! There is a magic method on all classes with `has_many` associations called `*collection*_count`, just like the counter cache. It will return a correct count value if you don't have a counter cache option set or the counter cache column value is null!

`:include`

Takes a list of second-order association names that should be eager-loaded when this object is loaded. A `SELECT` statement with the necessary `LEFT OUTER JOINS` will be constructed on the fly so that all the data needed to construct a whole object graph is queried in one database request.

With judicious use of `:include` and careful benchmarking, you can sometimes improve the performance of your application dramatically, mostly by eliminating N+1 queries. On the other hand, since doing huge multijoin queries and instantiating large object trees can also get very costly, certain usages of `:include` can actually make your application perform much more slowly. As they say, *your mileage may vary.*

Wilson Says...

If `:include` speeds your app up, it's too complicated and you should redesign it.

`:polymorphic => true`

Use the `:polymorphic` option to specify that an object is related to its association in a *polymorphic* way, which is the Rails way of saying that the type of the related object is stored in the database along with its foreign key. By making a `belongs_to` relationship polymorphic, you abstract out the association so that any other model in the system can fill it.

Polymorphic associations let you trade some measure of relational integrity for the convenience of implementation in child relationships that are reused across your application. Common examples are models such as photo attachments, comments, notes, line items, and so on.

Let's illustrate by writing a `Comment` class that attaches to its subjects polymorphically. We'll associate it to both expense reports and timesheets. Listing 7.2 has the schema information in migration code, followed by the code for the classes involved.

Notice the :subject_type column, which stores the class name of the associated class.

Listing 7.2 Comment Class Using Polymorphic belongs_to Relationship

```
create_table :comments do |t|
  t.column :subject,      :string
  t.column :body,         :text
  t.column :subject_id,   :integer
  t.column :subject_type, :string
  t.column :created_at,   :datetime
end

class Comment < ActiveRecord::Base
  belongs_to :subject, :polymorphic => true
end

class ExpenseReport < ActiveRecord::Base
  belongs_to :user
  has_many :comments, :as => :subject
end

class Timesheet < ActiveRecord::Base
  belongs_to :user
  has_many :comments, :as => :subject
end
```

As you can see in the ExpenseReport and Timesheet classes of Listing 7.2, there is a corresponding syntax where you give ActiveRecord a clue that the relationship is polymorphic by specifying :as => :subject. We haven't even covered has_many relationships yet, and polymorphic relationships have their own section in Chapter 9. So before we get any further ahead of ourselves, let's take a look at has_many relationships.

The has_many Association

Just like it sounds, the has_many association allows you to define a relationship in which one model *has many* other models that *belong to* it. The sheer readability of code constructs such as has_many is a major reason that people fall in love with Rails.

The `has_many` class method is often used without additional options. If Rails can guess the type of class in the relationship from the name of the association, no additional configuration is necessary. This bit of code should look familiar by now:

```
class User
  has_many :timesheets
  has_many :expense_reports
```

The names of the associations can be singularized and match the names of models in the application, so everything works as expected.

`has_many` Options

Despite the ease of use of `has_many`, there is a surprising amount of power and customization possible for those who know and understand the options available.

`:after_add`

Called after a record is added to the collection via the << method. Is not triggered by the collection's `create` method, so careful consideration is needed when relying on association callbacks.

Add callback method options to a `has_many` by passing one or more symbols corresponding to method names, or `Proc` objects. See Listing 7.3 in the `:before_add` option for an example.

`:after_remove`

Called after a record has been removed from the collection with the `delete` method. Add callback method options to a `has_many` by passing one or more symbols corresponding to method names, or `Proc` objects. See Listing 7.3 in the `:before_add` option for an example.

`:as`

Specifies the polymorphic `belongs_to` association to use on the related class. (See Chapter 9 for more about polymorphic relationships.)

:before_add

Triggered when a record is added to the collection via the << method. (Remember that concat and push are aliases of <<.) Raising an exception in the callback will stop the object from getting added to the collection. (Basically, because the callback is triggered right after the type mismatch check, and there is no rescue clause to be found inside <<.)

Add callback method options to a has_many by passing one or more symbols corresponding to method names, or Proc objects. You can set the option to either a single callback (as a Symbol or Proc) or to an array of them.

Listing 7.3 A Simple Example of :before_add Callback Usage

```
has_many :unchangable_posts,
         :class_name => "Post",
         :before_add => :raise_exception

private

  def raise_exception(object)
    raise "You can't add a post"
  end
```

Of course, that would have been a lot shorter code using a Proc since it's a one liner. The owner parameter is the object with the association. The record parameter is the object being added.

```
has_many :unchangable_posts,
  :class_name => "Post",
  :before_add => Proc.new {|owner, record| raise "Can't do it!"}
```

One more time, with a lambda, which doesn't check the *arity* of block parameters:

```
has_many :unchangable_posts,
  :class_name => "Post",
  :before_add => lamda {raise "You can't add a post"}
```

:before_remove

Called before a record is removed from a collection with the delete method. See before_add for more information.

:class_name

The :class_name option is common to all of the associations. It allows you to specify, as a string, the name of the class of the association, and is needed when the class name cannot be inferred from the name of the association itself.

:conditions

The :conditions option is common to all of the associations. It allows you to add extra conditions to the ActiveRecord-generated SQL query that bring back the objects in the association.

You can apply extra :conditions to an association for a variety of reasons. How about approval?

```
has_many :comments, :conditions => ['approved = ?', true]
```

Plus, there's no rule that you can't have more than one has_many association exposing the same two related tables in different ways. Just remember that you'll probably have to specify the class name too.

```
has_many :pending_comments, :conditions => ['approved = ?', true],
                            :class_name => 'Comment'
```

:counter_sql

Overrides the ActiveRecord-generated SQL query that would be used to count the number of records belonging to this association. Not necessarily needed in conjunction with the :finder_sql option, since ActiveRecord will automatically generate counter SQL code based on the custom finder SQL statement.

As with all custom SQL specifications in ActiveRecord, you must use single-quotes around the entire string to prevent premature interpolation. (That is, you don't want the string to get interpolated in the context of the class where you're declaring the association. You want it to get interpolated at runtime.)

```
has_many :things, :finder_sql => 'select * from t where id = #{id}'
```

:delete_sql

Overrides the ActiveRecord-generated SQL statement that would be used to break associations. Access to the associated model is provided via the record method.