Addison-Wesley Professional Ruby Series

THE RAILS 3 WAY

Foreword by **David H. Hansson**
creator of Ruby on Rails

OBIE FERNANDEZ

Praise for the Previous Edition

This encyclopedic book is not only a definitive Rails reference, but an indispensable guide to Software-as-a-Service coding techniques for serious craftspersons. I keep a copy in the lab, a copy at home, and a copy on each of my three e-book readers, and it's on the short list of essential resources for my undergraduate software engineering course.

—Armando Fox, adjunct associate professor, University of California, Berkeley

Everyone interested in Rails, at some point, has to follow *The Rails Way*.

—Fabio Cevasco, senior technical writer, Siemens AG, and blogger at H3RALD.com

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 our a pro on the other side.

—Dirk Elmendorf, co-founder of Rackspace, and 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

CHAPTER 7
Active Record Associations

> Any time you can reify something, you can create something that embodies a concept, it gives you leverage to work with it more powerfully. That's exactly what's going on with `has_many:through`.
>
> —Josh Susser

Active Record associations let you declaratively express relationships between model classes. The power and readability of the Associations API is an important part of what makes working with Rails so special.

This chapter covers the different kinds of Active Record associations available while highlighting use cases and available customizations for each of them. We also take a look at the classes that give us access to relationships themselves.

7.1 The Association Hierarchy

Associations typically appear as methods on Active Record model objects. For example, the method `timesheets` might represent the timesheets associated with a given `user`.

```
user.timesheets
```

However, people might get confused about the type of objects that are returned by association with these methods. This is because they have a way of masquerading as plain old Ruby objects and arrays (depending on the type of association we're considering). In the snippet, the `timesheets` method may appear to return an array of project objects.

The console will even confirm our thoughts. Ask any association collection what its return type is and it will tell you that it is an `Array`:

```
>> obie.timesheets.class
=> Array
```

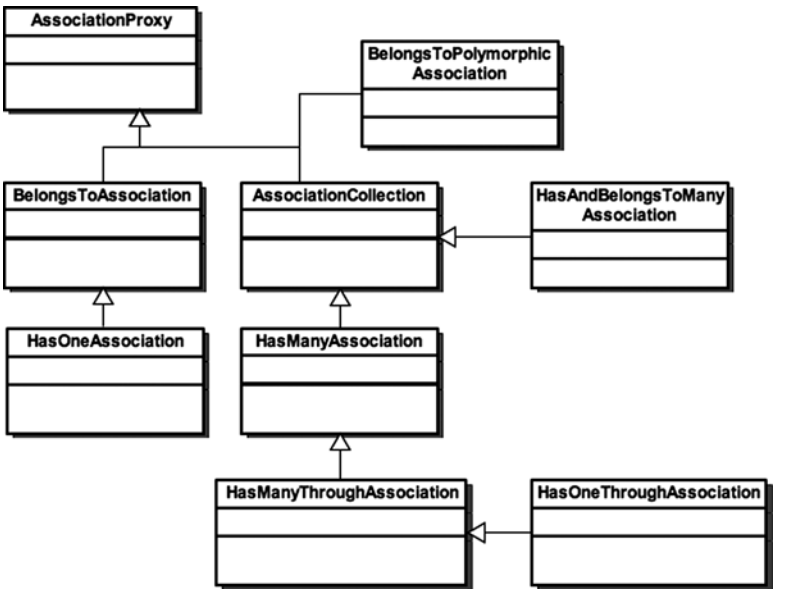

Figure 7.1 The Association proxies in their class hierarchy.

It's actually lying to you, albeit very innocently. Association methods for `has_many` associations are actually instances of `HasManyAssociation`, shown within its class hierarchy in Figure 7.1.

The parent class of all associations is `AssociationProxy`. It contains the basic structure and functionality of all assocation proxies. If you look near the top of its source code excerpted in Listing 7.1, you'll notice that it undefines a bunch of methods.

Listing 7.1 Excerpt from **`lib/active_record/associations/association_proxy.rb`**

```
instance_methods.each do |m|
  undef_method m unless m =~ /^(?:nil?|send|object_id|to_a)$|^__|proxy_/
end
```

As a result, most normal instance methods aren't actually defined on the proxy anymore, but are instead delegated to the target of the proxy via `method_missing`. That means that a call to `timesheets.class` returns the class of the underlying array rather than

the proxy. You can prove that `timesheet` is actually a proxy by asking it if it responds to one of `AssociationProxy`'s public methods, such as `proxy_owner`:

```
>> obie.timesheets.respond_to? :proxy_owner
=> true
```

Fortunately, it's not the Ruby way to care about the actual class of an object. What messages an object responds to is a lot more significant.

The parent class of all `has_many` associations is AssociationCollection and most of the methods that it defines work similarly regardless of the options declared for the relationship. Before we get much further into the details of the association proxies, let's delve into the most fundamental type of association that is commonly used in Rails applications: the `has_many` / `belongs_to` pair, used to define one-to-many relationships.

7.2 One-to-Many Relationships

In our recurring sample application, an example of a one-to-many relationship is the association between the `User`, `Timesheet`, and `ExpenseReport` classes:

```
class User < ActiveRecord::Base
  has_many :timesheets
  has_many :expense_reports
end
```

Timesheets and expense reports should be linked in the opposite direction as well, so that it is possible to reference the `user` to which a timesheet or expense report belongs.

```
class Timesheet < ActiveRecord::Base
  belongs_to :user
end

class ExpenseReport < ActiveRecord::Base
  belongs_to :user
end
```

When these relationship declarations are executed, Rails uses some metaprogramming magic to dynamically add code to your models. In particular, proxy collection objects are created that let you manipulate the relationship easily. To demonstrate, let's play with these relationships in the console. First, I'll create a user.

```
>> obie = User.create :login => 'obie', :password => '1234',
:password_confirmation => '1234', :email => 'obiefernandez@gmail.com'
=> #<User...>
```

Now I'll verify that I have collections for timesheets and expense reports.

```
>> obie.timesheets
ActiveRecord::StatementInvalid: SQLite3::SQLException: no such column:
timesheets.user_id:
SELECT * FROM timesheets WHERE (timesheets.user_id = 1)
from /.../connection_adapters/abstract_adapter.rb:128:in `log'
```

As David might say, "Whoops!" I forgot to add the foreign key columns to the `timesheets` and `expense_reports` tables, so in order to go forward I'll generate a migration for the changes:

```
$ rails generate migration add_user_foreign_keys
      exists       db/migrate
      create       db/migrate/20100108014048_add_user_foreign_keys.rb
```

Then I'll open `db/migrate/20100108014048_add_user_foreign_keys.rb` and add the missing columns. (Using `change_table` would mean writing many more lines of code, so we'll stick with the traditional `add_column` syntax, which still works fine.)

```
class AddUserForeignKeys < ActiveRecord::Migration
  def self.up
    add_column :timesheets, :user_id, :integer
    add_column :expense_reports, :user_id, :integer
  end

  def self.down
    remove_column :timesheets, :user_id
    remove_column :expense_reports, :user_id
  end
end
```

Running `rake db:migrate` applies the changes:

```
$ rake db:migrate
(in /Users/obie/prorails/time_and_expenses)
== AddUserForeignKeys: migrating
==============================================
-- add_column(:timesheets, :user_id, :integer)
   -> 0.0253s
-- add_column(:expense_reports, :user_id, :integer)
   -> 0.0101s
== AddUserForeignKeys: migrated (0.0357s)
==============================================
```

Index associations for performance boost

Premature optimization is the root of all evil. Or something like that.[1] However, most experienced Rails developers don't mind adding indexes for foreign keys at the time that those are created. In the case of our migration example, you'd add the following statements.

```
add_index :timesheets, :user_id
add_index :expense_reports, :user_id
```

Loading of your associations (which is usually more common than creation of items) will get a big performance boost.

Now I should be able to add a new blank timesheet to my user and check `timesheets` again to make sure it's there:

```
>> obie = User.find(1)
=> #<User id: 1...>
>> obie.timesheets << Timesheet.new
=> [#<Timesheet id: 1, user_id: 1...>]
>> obie.timesheets
=> [#<Timesheet id: 1, user_id: 1...>]
```

Notice that the `Timesheet` object gains an `id` immediately.

7.2.1 Adding Associated Objects to a Collection

As you can deduce from the previous example, appending an object to a `has_many` collection automatically saves that object. That is, unless the parent object (the owner of the collection) is not yet stored in the database. Let's make sure that's the case using Active Record's `reload` method, which re-fetches the attributes of an object from the database:

```
>> obie.timesheets.reload
=> [#<Timesheet id: 1, user_id: 1...>]
```

There it is. The foreign key, `user_id`, was automatically set by the `<<` method. It takes one or more association objects to add to the collection, and since it flattens its argument list and inserts each record, `push` and `concat` behave identically.

1. See `http://www.acm.org/ubiquity/views/v7i24_fallacy.html`

In the blank timesheet example, I could have used the `create` method on the association proxy, and it would have worked essentially the same way:

```
>> obie.timesheets.create
=> #<Timesheet id: 1, user_id: 1...>
```

Even though at first glance << and `create` do the same thing, there are some important differences in how they're implemented that are covered in the following section.

7.2.2 Association Collection Methods

Association collections are basically fancy wrappers around a Ruby array, and have all of a normal array's methods. Named scopes and all of `ActiveRecord::Base`'s class methods are also available on association collections, including `find`, `order`, `where`, etc.

```
user.timesheets.where(:submitted => true).order('updated_at desc')
user.timesheets.late # assuming a scope :late defined on the Timesheet
class
```

The following methods of `AssociationCollection` are inherited by and available to association collections:

`<<(*records)` and `create(attributes = {})`

Both methods will add either a single associated object or many, depending on whether you pass them an array or not. However, << is transactional, and `create` is not.

Yet another difference has to do with association callbacks (covered in this chapter's options section for `has_many`). The << method triggers the `:before_add` and `:after_add` callbacks, but the `create` method does not.

Finally, the return value behavior of both methods varies wildly. The `create` method returns the new instance created, which is what you'd expect given its counterpart in `ActiveRecord::Base`. The << method returns the association proxy (ever masquerading as an array), which allows chaining and is also natural behavior for a Ruby array.

However, << will return `false` and not itself if any of the records being added causes the operation to fail. You shouldn't depend on the return value of << being an array that you can continue operating on in a chained fashion.

`all`

The `all` method exists here mostly for consistency, since normally if you wanted to operate on all records you would simply use the association itself.