**Active Record Associations**

**Foreign Keys and Class Names**

When you create an association, Rails makes two major assumptions -- first, that the class of the model your association points to is based directly off of the name of the association, and, second, that the foreign key in any belongs\_to relationship will be called yourassociationname\_id. Any time you go away from these defaults, you just need to let Rails know what kind of class to look for and which foreign key to use.

A very simple case would be a User who can create many Posts for a blog::

# app/models/user.rb

class User < ActiveRecord::Base

has\_many :posts

end

# app/models/post.rb

class Post < ActiveRecord::Base

belongs\_to :user

end

So you could ask the first user for all her posts with User.first.posts or the first post for its author user with Post.first.user. Rails knows to look for a foreign key called user\_id in the Posts table. If you ask for Post.first.user, Rails will look in the Users table for the User with the ID corresponding to the user\_id column in the Posts table. All is well in the world when your association names correspond directly to the names of your models and tables.

But what if you want to have two types of users that the post belongs to -- the "author" and the "editor"? In this case, you'll need two separate foreign keys in your Posts table, presumably one called author\_id and another called editor\_id. How do you tell Rails that each of these foreign keys actually point to a User (so it knows to look in the Users table for them)? Just specify the class your association should point to using the aptly named :class\_name option:

# app/models/post.rb

class Post < ActiveRecord::Base

belongs\_to :author, :class\_name => "User"

belongs\_to :editor, :class\_name => "User"

end

In this case, Rails will automatically look for the foreign key named after the association, e.g. author\_id or editor\_id, in the Posts table.

If you called the association something which didn't correspond exactly to what you'd named the foreign key in your table, you need to tell Rails that as well. This should be obvious if you think of this relationship from the side of the User. The User will have some posts for which she is an author and others for which she is an editor. You'll need to rename the association on the User's side as well to keep things crystal clear, for instance splitting up authored\_posts from edited\_posts.

But now Rails doesn't have the foggiest idea where to look and what to look for. By default, if you ask for User.first.authored\_posts it will go looking in the authored\_posts table for a foreign key called user\_id (neither of which exist). To get it pointing at the right table, we again need to specify the:class\_name and to get it using the correct foreign key, we need to specify the right :foreign\_key. For instance:

# app/models/user.rb

class User < ActiveRecord::Base

has\_many :authored\_posts, :foreign\_key => "author\_id", :class\_name => "Post"

has\_many :edited\_posts, :foreign\_key => "editor\_id", :class\_name => "Post"

end

The basic gist of this is simple -- assume that Rails is looking for the foreign key named after the association in the table also named after the association. If any of these is incorrect because of a creatively named association or foreign key, you'll need to specify. This is quite common to make your associations more legible.

**Source**

Now that it's clear you need to let Rails know when you've creatively named your associations or foreign keys, I should point out that there's one additional step required if you're using a creatively namedhas\_many :through association. Recall that has-many-through associations are where you create a "through table" to act as a go-between for two models that have a many-to-many relationship.

For example, perhaps we change the example above so a Post actually can have multiple Authors (but still only one editor). We'll need to create a new table, which we'll call post\_authorings. post\_authorings joins these two models together and contains columns for authored\_post\_id andpost\_author\_id. You can probably see where this is going -- we've named our foreign keys something more descriptive and helpful than just simplypost\_id and user\_id but it will require us to inform Rails of the change. Our models look like:

# app/models/post.rb

class Post < ActiveRecord::Base

has\_many :post\_authorings, :foreign\_key => :authored\_post\_id

has\_many :authors, :through => :post\_authorings, :source => :post\_author

belongs\_to :editor, :class\_name => "User"

end

# app/models/user.rb

class User < ActiveRecord::Base

has\_many :post\_authorings, :foreign\_key => :post\_author\_id

has\_many :authored\_posts, :through => :post\_authorings

has\_many :edited\_posts, :foreign\_key => :editor\_id, :class\_name => "Post"

end

# app/models/post\_authoring.rb

class PostAuthoring < ActiveRecord::Base

belongs\_to :post\_author, :class\_name => "User"

belongs\_to :authored\_post, :class\_name => "Post"

end

The major thing to note here is that with has-many-through associations, Rails uses *the name of the association in the through table* to determine which foreign key and table name to reach out to. If it's named anything irregular, you'll use the :source option to specify which association actually points where we'd like to go. You can think of:source as being just like :class\_name but for the associations in the "through table".

It may be helpful to illustrate what Rails is doing. In the example above, if you ask for Post.first.authors, Rails sort of "thinks" like this:

1. You just called for the "authors" association on the Post model
2. To get "authors", we first need to go through the post\_authoringsassociation to get there.
3. Once we're at the PostAuthoring model, to get to the author, we'll need to use the :belongs\_to association and it'll be called post\_author. We know this because we used the :source option. If we hadn't used the:source option in the original has-many-through association, we would have been looking for belongs\_to :author instead.
4. Now we've got all the information we need to structure our SQL joins and grab the list of authors for the post.

It sounds a bit wonky but it's just the same logic as before -- if Rails can't tell based on its assumptions which associations or class names or foreign keys you're supposed to use, you need to specify them yourself using :source or :foreign\_key or :class\_name. It takes some practice but you'll get it. Usually you know something's up if you get error messages of the flavor ActiveRecord::StatementInvalid: SQLite3::SQLException: no such column.

Once you do get things figured out, it can still be helpful to look in your Rails server output to see which joins are being done to build the SQL query. That's a great window into what your associations are doing behind the scenes (because in the end, it's all about figuring out the correct SQL query to run).

**Polymorphic Associations**

We'll cover polymorphism here but, if your head is really spinning from the other concepts, feel free to just skim it. Consider this an "additional topic" instead of a core topic like foreign keys, class names and source.

Polymorphic associations can be a bit of a head scratcher at first and aren't terribly common, but are well suited for their use case. They use a big word to describe a pretty straightforward concept -- what if you have a single model that can belong to a bunch of different types of models? For example, let's say you're building a service like Facebook where users can comment on any of the different types of things posted by other users (like text, pictures, images). How do you make it okay to comment on all these different types of objects using just a single Comment model?

In a plain vanilla situation, the comment would belongs\_to a Post or a Picture or a Video (or whatever you're commenting on). You would have a foreign key called something like post\_id in your Comments table. Now if we want to be able to comment on multiple types of things, we need to figure out a different way of dealing with the foreign key because a single foreign key could be referencing a post, an image, a video etc and we don't know which one... it's ambiguous. You could just make a different column for each one, e.g. post\_id, image\_id, video\_id, but that is terribly inelegant and hardcoded (imagine if there were 100 different types of posts we want to be able to comment on!). We need to stick with a single foreign key column.

We solve this by storing not just the foreign key **id**, but also a reference to which **type** of model it corresponds to. That way, whenever you want to retrieve a comment, by specifying which type of thing it belongs to it is no longer ambiguous what you're asking for. Note that Rails does this for you in the background as long as it knows you're working with a polymorphic association.

We have to call our foreign key something a bit different from the normal case since it's ambiguous which model it's referencing and you can't just use post\_id or picture\_id. A convention is to come up with an abstract term for what type of action you're doing and use that to name the association. So in this case we're commenting on things and can thus call the foreign key "commentable". You'll see the \*able convention used a fair bit. So the migration for that model might look like:

class CreateComments < ActiveRecord::Migration

def change

create\_table :comments do |t|

t.string :title

t.text :content

t.integer :commentable\_id

t.string :commentable\_type

t.timestamps

end

end

end

"Commentable" will be used to refer to the associations as well. You'll need to tell your Comment model that it is actually polymorphic so Rails knows to also check for a commentable\_type column when using it. This is done simply:

# app/models/comments.rb

class Comment < ActiveRecord::Base

belongs\_to :commentable, :polymorphic => true

end

On the other side of the association, you just treat your comment like any other association (which happens to have a nonstandard name). You just need to specify the association on every model that has\_many of it. The only wrinkle is that, because it's using the "commentable" name, you need to specify it in an alias just like you would if any other association had a nonstandard name:

# app/models/post.rb

class Post < ActiveRecord::Base

has\_many :comments, :as => :commentable

end

# app/models/picture.rb

class Picture < ActiveRecord::Base

has\_many :comments, :as => :commentable

end

Rails does the rest of the work for you. Any time you ask a Picture for all its comments (Picture.first.comments), Rails will return just the comments that belong to that picture without you having to worry about anything else.

**Self Joins**

Often times you have relationships between the same type of model, for instance users who can follow other users. In this case, you need to specify both associations in your User model but name them differently. You will need to specify in your has\_many association what the name of the foreign\_key will be:

class Employee < ActiveRecord::Base

has\_many :subordinates, class\_name: "Employee",

foreign\_key: "manager\_id"

belongs\_to :manager, class\_name: "Employee"

end

**Creating Association Objects**

There's a couple of shortcuts for creating new association objects. The first is to call #new or #create on the association to automatically populate the foreign key. For instance, if a User has\_many Posts and Post belongs\_toa User:

# Long version:

> user = User.first

> post = Post.create(:title => "sample", :user\_id => user.id)

# Shorter version:

> user = User.first

> user.posts.create(:title => "sample")

Another nifty trick is to create a new object and then use the shovel operator to add it to the association. This is just one of the ways that**associations sometimes act like arrays**:

> user = User.create(:name => "foobar")

> post = Post.new(:title => "sample")

> user.posts << post

This will save the post to the database with that User's foreign key. It's roughly equivalent to calling:

> post.user\_id = user.id

> post.save

If you really want to, you can actually replace the entire association with a new collection by setting it equal to the new collection:

> user = User.first

> post1 = Post.find(1)

> post2 = Post.find(2)

> user.posts = [post1, post2] # posts added to that user's collection

**Destroying Dependents**

If your user has created a bunch of posts and then decides to delete her account, how do you delete all the associated posts? Specify the:dependent => :destroy option when first declaring the association:

# app/models/user.rb

class User < ActiveRecord::Base

has\_many :posts, :dependent => :destroy

end

This is just the most common among several options to specify for:dependent. It will run the destroy method on all objects that belong to that user when the user is destroyed.