Tutorial: ActiveRecord::Associations

Daniel Swain 02/05/2016

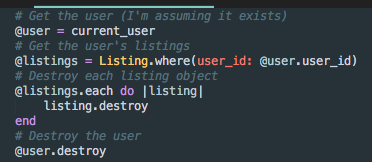
If you’ve been following Nichola’s tutorials up to and including Session 11 then you would have gotten familiar with ActiveRecord models and how to perform simple Create, Read, Update and Destroy (CRUD) actions.

These actions form the cornerstone of all your Rails application’s database methods, but they can be cumbersome to implement if you have models that relate to other models. An example for our Real Estate Project is if you want to get all the listings for a particular user you might use the following command in a controller or the console



Here, I’m getting the current logged in user and then looking in the database through a Listing model for all objects that have the same **user\_id** as the current user.

This works, and allows us to get all the listings for a given user, **but what happens if we delete that user? Would the listings for that user still exist?**

**The answer is yes**, we’d end up with orphaned listings that exist in the Listings table and refer to a user id which doesn’t exist. This could lead to application errors later if you try and use a user object retrieved through that listing (i.e. **user = User.find\_by\_id(listing.user\_id)** will return nil as the user with that id no longer exists). To delete the listings in this example, we’d need an extra step where we find all listings with the user id of the user deleted and then delete them, necessitating an extra loop: 

The solution to this problem is through the use of the ActiveRecord::Associations class. As we’ll see later, this skips looping through each listing and handles the dependency actions for us elegantly (as long as you set it up correctly).

ActiveRecord::Associations (Associations from now on) allow us to declare associations between ActiveRecord Models and retrieve associated models and perform actions on associated records. They solve the problem identified above where we have orphaned data after deleting a user and provide simple helper accessors for getting associated objects for a given object.

To simplify this, if you’ve set up your associations correctly we can modify the code above to look like the following:



Where **user\_listing** is the name of the association we defined that retrieves all the listings that the user object has.

A nice feature of Associations is you can set dependent actions and at the end of this tutorial, if you were to delete the user object retrieved in the code above, then all the listing objects relating to that user would also be deleted from the database.

# Types of Associations

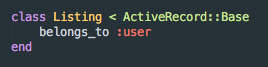
There are four main associations available in your Rails application. They are **belongs\_to, has\_one, has\_many** and **has\_and\_belongs\_to\_many**. These represent the common **one-to-one, one-to-many** and **many-to-many** relationships that you may be familiar with from other frameworks or database relationship schemas.

The associations types are discussed in more detail below:

## belongs\_to

A **belongs\_to** association can be thought of as a **One to One** relationship (i.e. only one object relating to a single different object) where each object of the Model class this is defined in **belongs to** the referenced Model Class object.

In our example below this is declaring that a Listing **belongs to** a single User object.



**Note, for this and all associations**, the first symbol after the association type is the name of the association and is how you refer to the association on the object. So in the association above we would get the user the listing belongs to by entering in **user = listing.user**.

## has\_one

A **has\_one** association is similar to a **belongs\_to** association in that it is a **One to One** relationship, however the actual database saving is handled differently. What this means is that if I say **Object1 has\_one** of **Object2**, then **Object2** actually saves the id of the **Object1** that **has\_one** of itself in its own table.

In our example, if I say a Cover Image **has\_one** Listing. Then I would need a column in the Listing table to store the single cover image id. This is hopefully described in more detail below.

This concept is a bit difficult and not as commonly used as **belongs\_to** and **has\_many**. But If I have a listing that has a single cover image (and a Model Class CoverImage) and I’m saving the cover image in the listing database and I only want the listing to have one cover image then I can define it in the CoverImage model using a **has\_one** association and define the opposite **belongs\_to** association in the Listing class (or the other side of the **has\_one** association).

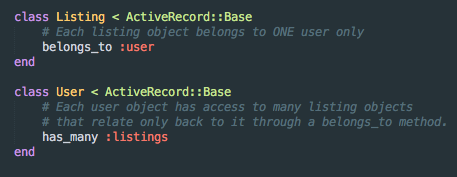


## has\_many

A has\_many association can be thought of as a One to Many relationship, where one object has relations with many other objects. This is useful for our project in declaring that a single user can have many listings.

Generally, the has\_many association is the other side of a belongs\_to association as we’re saying Object1 can relate to many Object2’s but Object2 belongs only to Object1. This is useful to declare both sides as we can get an instance of Object 1 from Object2 and get all Object2’s from an instance of Object1 (as an example).

See below for a simple example declaration:

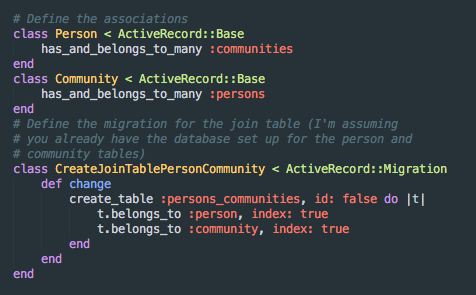


## has\_and\_belongs\_to\_many

A **has\_and\_belongs\_to\_many** association can be thought of as a **Many to Many** relationship, where one object can both **belong to**, and **have many** of another relationship. This is a complex association and generally requires a separate table to store the pairings of each objects.

An example would be a situation where we have a table of People and a table of Communities. We would say that a Person can have and belong to many Communities, and that a Community can have and belong to many People. We would then need to store these pairings in a Join table of People\_Community pairings, storing the ids of the Person and their Community associations and vice versa.

See below for an example declaration:



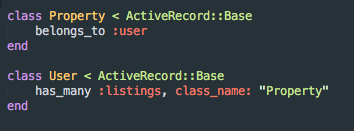
# Association options:

Each association defined above has various configuration options available to it on declaration, but the common methods I’ll discuss are the following:

## :class\_name

The **class\_name** is the name of the model class you’re associating to in the association. This is useful if ActiveRecord cannot determine the class name of your model from the association’s name.

i.e. if you have an association for your listings with your users, but your listings are actually defined by a model called Property then you might use the following:



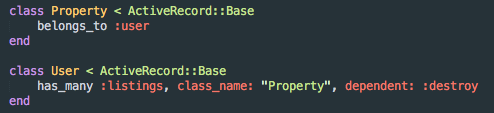
Here we have a user that would get all its listings using **user.listings** but we don’t have a Listings class, as we store our listings as a **Property** model object, so we need to tell our association that when getting the listings for the user it should look at the **Property** class for our model definition.

## :dependent

The dependent option tells Rails what to do to the model objects that are dependencies of the relationship you’re defining when you **delete** or **destroy** the defining object.

That sounds a bit confusing, and it confused me for a bit. But basically, if you are defining a relation in a Model Class and you give it the **dependent: :delete** option then if you are saying this **object 1** has\_many of another **object 2**, when you delete **object 1**, it will delete all **object 2’s** that are associated with it.

This is useful in our situation where a user being deleted would remove all the property listings associated with it. Extending the example above, setting the **dependent** option would look like this:



This image shows how we solve the problem highlighted at the beginning of this tutorial. Now, when we destroy our user object then the destroy method will be called on all associated Property (listing) objects, thereby removing them from the database.

**Points to note.**

* Don’t set the **dependent** option on a **belongs\_to** association where a **has\_many** exists on the other side as this can lead to you having orphaned data. The best practice is to set it in your **has\_many** associations.
* **THIS ONE IS IMPORTANT:** It’s best practice to use **:destroy** rather than **:delete**, as the **:destroy** method call will continue down all the relation/association paths, whereas **:delete** will just delete the associated object from the database.

What this means is that when I **:destroy** my user, I also call the **:destroy** method on each of its listings. When the **:destroy** method is called on each listing it will call any **:destroy** methods on associations defined in my listing model and so on and so on.

So if our listings have many images that we want to delete then by using the **:destroy** method the following will happen.

User object **:destroy** method called

* Each **dependent** listing object has their **:destroy** method called
  + Each listing’s own associations have their **:destroy** methods called

In this case deleting all the images for each listing.

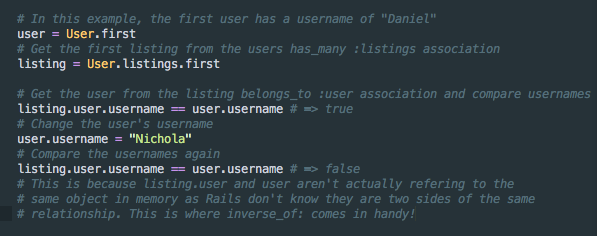
This is best practice as it will ensure we minimise the risk of orphaned data.

## :inverse\_of

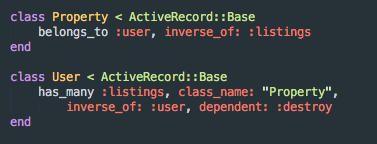
The **:inverse\_of** option tells rails that the association you’ve defined can be considered the direct opposite, or inverse, of the association referenced.

It is good practice to declare this option as Rails doesn’t automatically know whether associations are related. This means, in our examples above, if we got the first user object, and then the first listing object for that user and then edited the user object, the two objects would be out of sync as Rails doesn’t know about the connection between these two objects automatically. Rails would not update the user associated with the first listing object even though it’s the same user.

The following screenshot demonstrates the example.



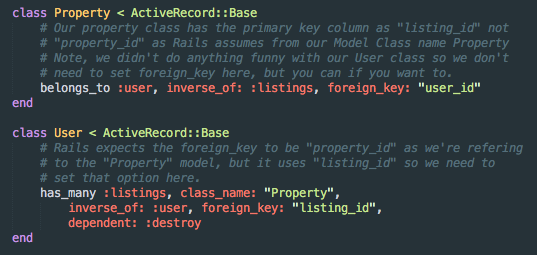
This can be fixed by declaring the **:inverse\_of** option when defining your association. To extend the example we have above we would add the **:inverse\_of** declaration as follows:



Notice that we use the name of the opposite association as a symbol. This is something to remember if you have a lot of associations with the User Model, it is best to use unique association names like **user\_listings**, or **user\_images** rather than **listings** and **images** as you might need those names again. Also, on the other side of the relationship we should probably use **property\_user** or **listing\_user** rather than **user** as we’re likely going to have multiple associations with the user object from other Model classes and don’t want any conflicts.

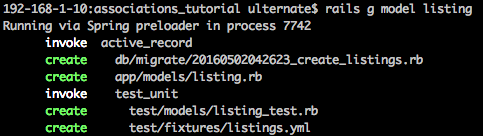
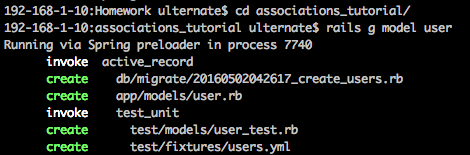
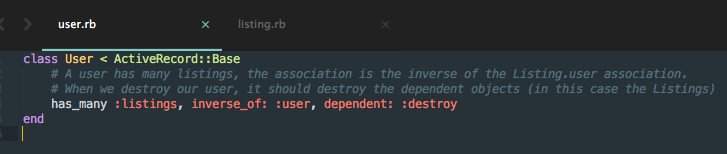
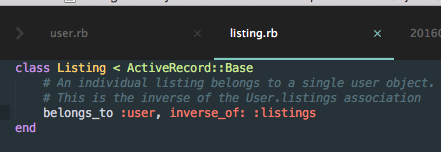
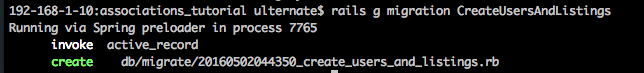
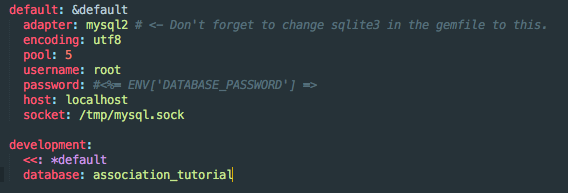
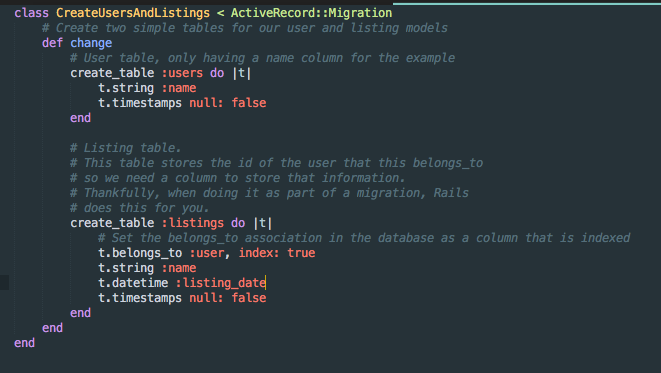
## :foreign\_key

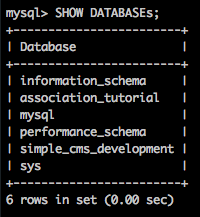
Similar to **class\_name**, **foreign\_key** is used where Rails can’t identify automatically what the column name is that we are referring to for the associated model objects id column. Rails automatically adds “**\_id**” to the associations model class name and assumes that is the primary key column for the model you’re associating with. If you’re primary key column is named differently then you should set the **foreign\_key** option to reflect the correct column name. See below for an example:

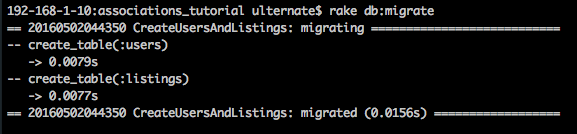
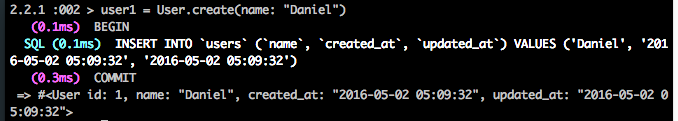
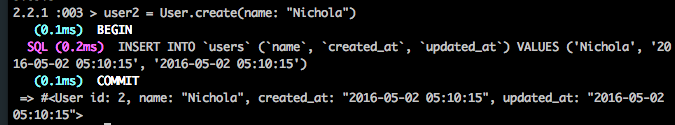
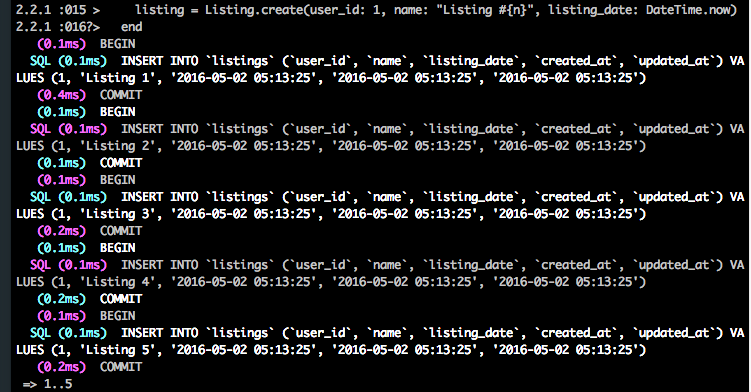
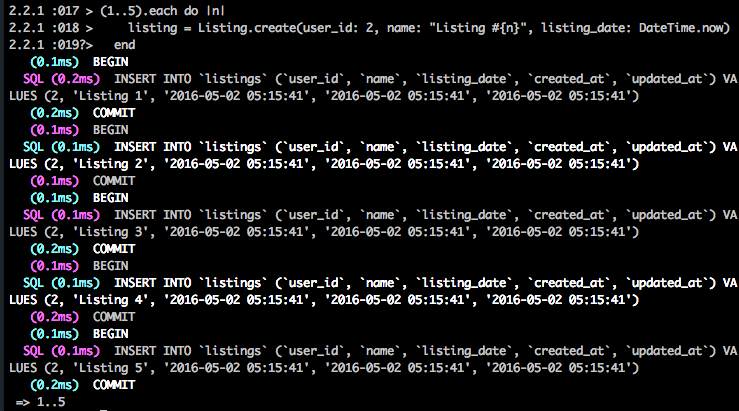
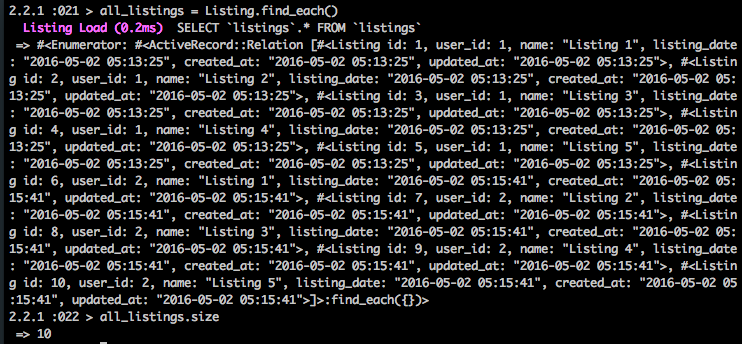
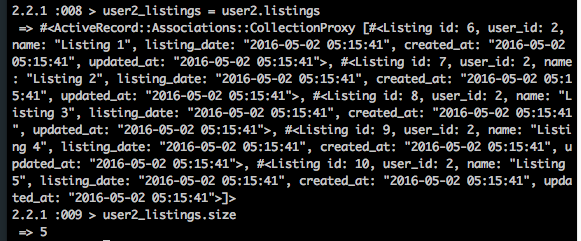
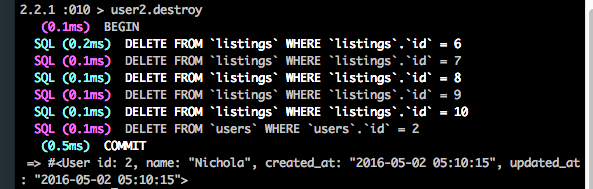
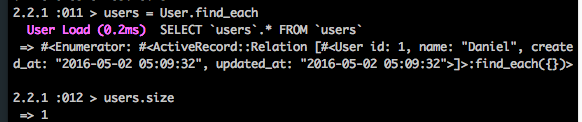
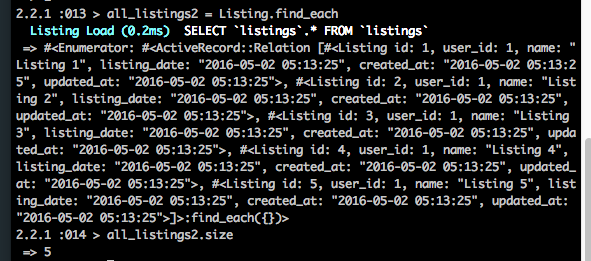


# TUTORIAL EXERCIES

Now you have a good idea of ActiveRecord::Associations we’re going to create a simple test app with a User Model, a Listing Model and use the **has\_many** and **belongs\_to** associations.

1. Open up your command prompt with Rails (or terminal, or bash, or whatever you’re using to use Rails in the console)
2. Change your directory to where you want to save the project (cd /path/to/directory)
3. Create your Rails app: 
4. Change directory into the newly created Rails application directory (in this case associations\_tutorial)
5. We need a User Model and a Listing Model so let’s create both of those
6. Now we have our models we need to add the associations to the objects, let’s do this in our model definition and migration for the User model first. We want to tell Rails that a User has\_many listings and that this will be the inverse of the belongs\_to user definition we’ll do in a bit for the Listing Model.
7. Now we do the same in the Listings.rb file but this time we use the belongs\_to association type.
8. Delete the two automatically generated migrations in **db/migrate** and create a new one called “CreateUsersAndListings”
9. Connect your rails app to MySQL using the guides from the Simple\_CMS tutorial (I’ve attached a screenshot of my config/database.yml file for guide)
10. Change ‘sqlite3’ to ‘mysql2’ in your Gemfile and run **bundle install**
11. Now, We’re actually going to edit the generated migration to create the two tables and set up the columns to store the association information in (in this case it’s stored in the listings table)
12. Before we can run the migration we need to create the database in MySQL. Login to MySQL using **mysql –u root –p** and then enter your password.
13. Create the database using **CREATE DATABASE association\_tutorial;**
14. Confirm by typing **SHOW DATABASES;**



1. Now we have a database to connect to run the migration using **rake db:migrate**
2. We’re now going to test out our database in the rails console. Type **rails c** to enter the console.
3. To test our association, we’re going to do the following:
   1. Create a user object **user1 = User.create(name: “Daniel”)**
   2. Create a second user object **user2 = User.create(name: “Nichola”)**
   3. In (d) and (e) you’ll see how I create a few listings for each user in **irb** using a simple each loop (which you can use in **irb**, just type each line, **irb** detects it’s a loop and automatically indents the next line. You must end the loop with the end value before the loop will run).
   4. Create a few listings for user1. (note, the screenshot misses the **(1..5).each do |n|** command I entered in first, see the screenshot for user2 for a better example
   5. Create a few listings for user2.
   6. View all the listings **all\_listings = Listing.find\_each()** We expect 10 listings.
   7. Get all the listings for user2, **user2\_listings = user2.listings** we expect 5 listings.****
   8. Delete user2, **user2.destroy()** If you look at the image below you’ll see that Rails went and deleted the listings from the Listings table that were associated with the User object 2 (awesome, no need to manually delete them all)
   9. Check to see if the user has been deleted by entering **users = User.find\_each**, this should have size = 1
   10. Get all listings again and should see that the ones from user2 have been removed, **all\_listings2 = Listing.find\_each()**

You’ve done it! Now you have a simple Rails application with a User model and a Listing model with an association setup between the user and the listings (**has\_many**) and each listing and a user (using **belongs\_to**). You’re also able to refer to each object from an instance of the other Model and deleting your User will delete all listings associated with that user!!!

Hopefully this helps you understand ActiveRecord::Associations and how they can help you in your Rails projects, particularly in the current Real Estate Website we’re all working on.

If you have any questions, don’t hesitate to ask. If you have any feedback, or spot any errors, please let me know.

For information straight from the horse’s mouth, have a look here: <http://guides.rubyonrails.org/association_basics.html>

Cheers,

Daniel