3

MODELS

In Rails, models represent the data in your application and the rules to manipulate that data. Models manage interactions between your application and a corresponding database table. The bulk of your application's business logic should also be in the models.

This chapter covers Active Record, the Rails component that provides model persistence (that is, storing data in the database), as well as data validations, database migrations, and model associations. *Validations* are rules to ensure that only valid data is stored in the database. You create database *migrations* to change the schema of the database, and *associations* are relationships between multiple models in your application.

The Post Model

In the previous chapter, we used the Rails scaffold generator to build a simple blog with models, views, and controllers for blog posts. Look at the post model created by the scaffold generator by opening the file *app/models/post.rb* in your favorite text editor.

```
class Post < ActiveRecord::Base
end
```

There's not much to see here. Right now, the file just tells us that the class `Post` inherits from `ActiveRecord::Base`. Before I talk about what you can actually do with `Post`, let's begin our discussion with Active Record.

Active Record

Active Record is an implementation of the object-relational mapping (ORM) pattern described, using the same name, by Martin Fowler in *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002). It's an automated mapping between classes and tables as well as attributes and columns.

Each table in your database is represented by a class in your application. Each row of that table is represented by an instance (or object) of the associated class, and each column of that row is represented by an attribute of that object. The example in Table 3-1 demonstrates this structure. If you could look inside your database, this is what you would see.

Table 3-1: The Posts Table

id	title	body	created_at	updated_at
1	Hello, World	Welcome to my blog...	...	...
2	My Cat	The cutest kitty in the...	...	...
3	Too Busy	Sorry I haven't posted...	...	...

Table 3-1 holds three example blog posts. This table is represented by the `Post` class. The post with an `id` of 1 can be represented by a `Post` object. Let's call our object `post`.

You can access the data associated with a single column by calling an attribute method on the object. For example, to see the post's title, call `post.title`. The ability to access and change database values by calling attribute methods on an object is known as *direct manipulation*.

Create, Read, Update, and Delete

Let's explore Active Record further by entering a few commands in the Rails console. The Rails console is the IRB that you used in Chapter 1 with your Rails application's environment preloaded.

To start the Rails console, go to your *blog* directory and enter **`bin/rails console`**. You might notice that the console takes a little longer to start than the IRB. During that slight pause, your application's environment is being loaded.

As with the IRB, you can enter `exit` to quit the console when you're done.

The four major functions of database applications are *create*, *read*, *update*, and *delete*, usually abbreviated as *CRUD*. Once you know how to perform these four actions, you can build any type of application you need.

Rails makes these actions easy for you. In most cases, you can accomplish each with a single line of code. Let's use them now to work with posts on our blog.

Create

We'll start by adding a few records to the database. Enter these commands in the Rails console as you work through this section. The remaining examples in this chapter use these records.

The easiest way to create a record in Rails is with the appropriately named `create` method, as shown here:

```
2.1.0 :001 > Post.create title: "First Post"
❶   (0.1ms) begin transaction
  SQL (0.4ms) INSERT INTO "posts" ("created_at"...
   (1.9ms) commit transaction
 => #<Post id: 1, title: "First Post", ...>
```

The Rails console displays the SQL being sent to the database as commands are run ❶. In the interest of brevity, I'm going to omit these SQL statements in the rest of the samples.

The `create` method accepts a hash of attribute-value pairs and inserts a record into the database with the appropriate values. In this case, it's setting the `title` attribute to the value `"First Post"`. When you run this example, the values for `id`, `created_at`, and `updated_at` are set for you automatically. The `id` column is an auto-incrementing value in the database, whereas `created_at` and `updated_at` are timestamps set for you by Rails. The `body` column is set to NULL since no value was passed for it.

The `create` method is a shortcut for instantiating a new `Post` object, assigning values, and saving it to the database. If you don't want to take the shortcut, you could also write a separate line of code for each action:

```
2.1.0 :002 > post = Post.new
 => #<Post id: nil, title: nil, ...>
2.1.0 :003 > post.title = "Second Post"
 => "Second Post"
2.1.0 :004 > post.save
 => true
```

We had to use multiple commands this time, but just like before, we've created a brand new `Post` object. Two posts are now stored in the database. In both examples, we only assigned values to the post's `title` attribute, but you would assign values to the post `body` in exactly the same way. Rails assigns values to `id`, `created_at`, and `updated_at` automatically. You shouldn't change these.

Read

Once you have a few posts in your database, you'll probably want to read them back out for display. First, let's look at all of the posts in the database with the `all` method:

```
2.1.0 :005 > posts = Post.all
 => #<ActiveRecord::Relation [#<Post id: 1, ...>, #<Post id: 2, ...>]>
```

This returns an Active Record *relation*, which contains an array of all posts in your database, and stores it in `posts`. You can chain additional methods onto this relation, and Active Record combines them into a single query.

Active Record also implements the `first` and `last` methods, which return the first and last entries in an array. The Active Record version of these methods returns only the first or last record in the database table. This is much more efficient than fetching all of the records in the table and then calling `first` or `last` on the array. Let's try fetching a couple of posts from our database:

```
2.1.0 :006 > Post.first
 => #<Post id: 1, title: "First Post", ...>
2.1.0 :007 > Post.last
 => #<Post id: 2, title: "Second Post", ...>
```

This example returns the first and last posts, as ordered by `id`. You'll learn how to order records by a different field in the next section. Sometimes, however, you'll know exactly which record you want, and it might not be the first or last one. In that case, you can use the `find` method to retrieve a record by `id`.

```
2.1.0 :008 > post = Post.find 2
 => #<Post id: 2, title: "Second Post", ...>
```

Just don't ask `find` to fetch a record that doesn't exist. If a record with the specified `id` isn't in your database, Active Record will raise an `ActiveRecord::RecordNotFound` exception. When you know a specific record exists but you don't know its `id`, you can use the `where` method to specify an attribute that you do know:

```
2.1.0 :009 > post = Post.where(title: "First Post").first
 => #<Post id: 1, title: "First Post", ...>
```

The `where` method also returns a relation. If more than one record matches, you can chain the `all` method after `where` and tell Rails to retrieve all matching records on demand when they are needed.

If you know the database has only one matching record, you can chain the `first` method after `where` to retrieve that specific record as in the previous example. This pattern is so common that Active Record also provides the `find_by` method as a shortcut:

```
2.1.0 :010 > post = Post.find_by title: "First Post"
 => #<Post id: 1, title: "First Post", ...>
```

This method takes a hash of attribute-value pairs and returns the first matching record.

Update

Updating a record is as easy as reading it into a variable, changing values via direct manipulation, and then saving it back to the database:

```
2.1.0 :011 > post = Post.find 2
 => #<Post id: 2, title: "Second Post", ...>
2.1.0 :012 > post.title = "2nd Post"
 => "2nd Post"
2.1.0 :013 > post.save
 => true
```

Rails also provides the `update` method, which takes a hash of attribute-value pairs, updates the record, and saves to the database all on one line:

```
2.1.0 :014 > post = Post.find 2
 => #<Post id: 2, title: "2nd Post", ...>
2.1.0 :015 > post.update title: "Second Post"
 => true
```

The `update` method, like the `save` method, returns `true` when successful or `false` if it has a problem saving the record.

Delete

Once you have read a record from the database, you can delete it with the `destroy` method. But this time don't type in these commands. You don't want to delete the posts you created earlier!

```
2.1.0 :016 > post = Post.find 2
 => #<Post id: 2, title: "Second Post", ...>
2.1.0 :017 > post.destroy
 => #<Post id: 2, title: "Second Post", ...>
```

The `destroy` method can also be called on the class to delete a record by `id`, which has the same effect as reading the record into a variable first:

```
2.1.0 :018 > Post.destroy 2
 => #<Post id: 2, title: "Second Post", ...>
```

You can also delete records based on a relation:

```
2.1.0 :019 > Post.where(title: "First Post").destroy_all
 => [#<Post id: 1, title: "First Post", ...>]
```

This example deletes all records with a title of `"First Post"`. Be careful with the `destroy_all` method, however. If you call it without a `where` clause, you'll delete all records of the specified class!

More Active Record Methods

If you're familiar with SQL or other methods of accessing records in a database, you know there's much more to working with a database than simple CRUD. Active Record provides methods for more database operations, such as ordering, limiting, counting, and other calculations.

Query Conditions

In addition to the simple `where` conditions you've seen so far, Active Record also has several methods to help refine your queries. The `order` method specifies the order of returned records; `limit` specifies how many records to return; and `offset` specifies the first record to return from a list.

The `limit` and `offset` methods are often used together for pagination. For example, if you want to show 10 blog posts per page, you can read the posts for the first page like this:

```
2.1.0 :020 > posts = Post.limit(10)
 => #<ActiveRecord::Relation [#<Post id: 1, ...>, #<Post id: 2, ...>]>
```

To read the posts for the second page of your site, you'll need to skip the first 10 posts:

```
2.1.0 :021 > posts = Post.limit(10).offset(10)
 => #<ActiveRecord::Relation []>
```

Entering this returns an empty set since we only have two posts in our database. When you combine `offset` with `limit` in this way, you can pass `offset` multiples of what you passed `limit` to view different pages of your blog.

You can also change how the entries in a relation are ordered. When using `limit`, the order of records returned is undefined, so you need to specify an order. With the `order` method, you can specify a different order for the set of records returned:

```
2.1.0 :022 > posts = Post.limit(10).order "created_at DESC"
 => #<ActiveRecord::Relation [#<Post id: 2, ...>, #<Post id: 1, ...>]>
```

Using `DESC` tells `order` to return the posts from newest to oldest. You could also use `ASC` to order them the opposite way. If you would rather see

posts alphabetized by title, try replacing `"created_at DESC"` with `"title ASC"`. The `order` method defaults to ascending order if you don't specify `ASC` or `DESC`, but I always give an order so my intention is clear.

Calculations

Databases also provide methods for performing calculations on records. We could read the records and perform these operations in Ruby, but the methods built in to the database are usually optimized to be faster and use less memory.

The `count` method returns the number of records matching a given condition:

```
2.1.0 :023 > count = Post.count
 => 2
```

If you don't specify a condition, `count` counts all records by default, as in this example.

The `sum`, `average`, `minimum`, and `maximum` methods perform the requested function on a field. For example, this line of code finds and returns the date on the newest blog post:

```
2.1.0 :024 > date = Post.maximum :created_at
 => 2014-03-12 04:10:08 UTC
```

The maximum `created_at` date you see should match the date for your newest blog post, not necessarily the date you see in the example.

Migrations

Database migrations are used any time you need to change your database's structure. When we used the scaffold generator to create blog posts, it generated a migration for us, but you can also create migrations yourself. As you build your application, your database migrations contain a complete record of the changes made to your database.

Migration files are stored in the *db/migrate* directory and start with a timestamp that indicates when they were created. For example, you can see the migration created by the scaffold generator by editing the file *db/migrate/*_create_posts.rb*. (Because the timestamps on your files will surely be different from mine, I'll use an asterisk from now on to refer to the date part of the filename.) Let's look at that file now:

```
class CreatePosts < ActiveRecord::Migration
❶  def change
     create_table :posts do |t|
       t.string :title
       t.text :body

       t.timestamps
```

```
    end
  end
end
```

Database migrations are actually Ruby classes. The `change` method is called ❶ when the migration is run. In this case, the method creates a table named `posts` with fields for `title`, `body`, and `timestamps`. The `timestamps` field refers to both the `created_at` and `updated_at` fields. Rails also automatically adds the `id` column.

You can run migrations as tasks with the `rake` command. For example, you enter `bin/rake db:migrate` to run all pending migrations and bring your database up-to-date.

Rails keeps track of which migrations have been run by storing the timestamps in a database table called `schema_migrations`.

If you make a mistake in a database migration, use the `db:rollback` task to undo it. After you correct the migration, use `db:migrate` to run it again.

The Schema

In addition to the individual migration files, Rails also stores your database's current state. You can see this by opening the file *db/schema.rb*. Ignoring the comment block at the top of the file, it should look like this:

```
--snip--
ActiveRecord::Schema.define(version: 20130523013959) do

  create_table "posts", force: true do |t|
    t.string   "title"
    t.text     "body"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

end
```

This file is updated whenever you run a database migration. You should not edit it manually. If you are moving your application to a new computer and would like to create a new, empty database all at once instead of by running the individual migrations, you can do that with the `db:schema:load` `rake` task:

```
$ bin/rake db:schema:load
```

Running this command resets the database structure and removes all of your data in the process.

Adding a Column

Now that you know more about migrations, let's create one and run it. When we created our blog post model, we forgot that posts need authors. Add a string column to the posts table by generating a new migration:

```
$ bin/rails g migration add_author_to_posts author:string
```

The Rails generator (`g` is short for `generate`) looks at the name of your migration, in this case, `add_author_to_posts`, and tries to figure out what you want to do. This is another example of convention over configuration: name your migration in the format `add_ColumnName_to_TableName`, and Rails will parse that to add what you need. Based on the name, we clearly want to add a column named `author` to the posts table. We also specified that `author` is a string, so Rails has all the information it needs to create the migration.

NOTE *You can name a migration anything you want, but you should follow the convention so you don't have to edit the migration manually.*

Enter **`bin/rake db:migrate`** to run the migration and add the `author` column to your database. If you still have a Rails console open, you'll need to **`exit`** and restart with **`bin/rails console`** for your changes to take effect. You can also look at the *db/schema.rb* file to see the new column in the posts table.

Inside the Author Migration

The code you just generated for adding a column is simple. Edit the file *db/migrate/*_add_author_to_posts.rb* to see how it works.

```
class AddAuthorToPosts < ActiveRecord::Migration
  def change
    add_column :posts, :author, :string
  end
end
```

Like *_*create_posts.rb*, this migration is a class containing a `change` method. The `add_column` method is called with the table name, column name, and column type. If you want to add multiple columns, you could create separate migrations for each, or you could call this method multiple times.

Active Record migrations also provide the `rename_column` method for changing a column's name, the `remove_column` method for removing a column from a table, and the `change_column` method for changing a column's type or other options, such as default value.

Validations

Remember that models have rules for manipulating application data. Active Record *validations* are sets of rules created to protect your data. Add validation rules to ensure that only good data makes it into your database.

Adding a Validation

Let's look at an example. Because we're making a blog, we should ensure that all posts have a title so readers don't get confused, and we can do that with a validation rule.

Validations are implemented as class methods in Rails. Open the post model (*app/models/post.rb*) in your editor and add this line:

```
class Post < ActiveRecord::Base
  validates :title, :presence => true
end
```

This validates the presence of text in the `title` field. Attempting to create a blog post with a blank title should now result in an error.

OTHER COMMON VALIDATIONS

Rails provides a variety of other validations in addition to the `:presence` validation. For example, you can use the `:uniqueness` validation to ensure that no two posts have the same title.

The `:length` validation accepts a hash of options to confirm that the value is the correct length. Adding this line to your post model confirms that all titles are at least five characters:

```
validates :title, :length => { :minimum => 5 }
```

You can also specify a `:maximum` value instead of a `:minimum`, or you can use `:is` to set an exact value.

The `:exclusion` validation ensures the value does not belong to a given set of values. For example, adding this validation prohibits blog posts with the title *Title*:

```
validates :title, :exclusion => { :in => [ "Title" ] }
```

You can think of `:exclusion` as a blacklist for values you don't want to allow. Rails also provides an `:inclusion` validation for specifying a whitelist of accepted values.

Testing Data

Validations are automatically run before data is saved to the database. Attempt to store invalid data, and `save` returns `false`. You can also test a model manually with the `valid?` method:

```
2.1.0 :025 > post = Post.new
 => #<Post id: nil, title: nil, ...>
2.1.0 :026 > post.valid?
 => false
2.1.0 :027 > post.errors.full_messages
 => ["Title can't be blank"]
```

In this example, the `valid?` method should return `false` because you didn't set a value for the title. Failing validations add messages to an array called `errors`, and calling `full_messages` on the `errors` array should return a list of error messages generated by Active Record based on your validations.

Use validations freely to keep bad data out of your database, but also consider your users when you create those validations. Make it clear which values are valid, and display error messages if invalid data is given so the user can correct the mistake.

Associations

Only the simplest of applications contain a single model. As your application grows, you'll need additional models, and as you add more, you'll need to describe the relationships between them. Active Record *associations* describe the relationships between models. For example, let's add comments to our blog posts.

Posts and comments are associated. Each post *has many* comments, and each comment *belongs to* a post. This *one-to-many* relationship is one of the most commonly used associations, and we'll explore it here.

Generating the Model

A blog comment should have an author, a body, and a reference to a post. You can easily generate a model using that information:

```
$ bin/rails g model Comment author:string body:text post:references
```

NOTE *Remember to run database migrations after generating this new model!*

The `post:references` option tells the Rails generator to add a foreign key to the comments database table. In this case, the foreign key is named `post_id` because it refers to a post. The `post_id` field contains the `id` of this comment's post. The migration created the column we need in the database, so now we need to edit our models to finish setting up the association.

Adding Associations

First, open *app/model/post.rb* again to add the comments association. Earlier I said that each post has many comments, and that's the association we need here:

```
class Post < ActiveRecord::Base
  validates :title, :presence => true
  has_many :comments
end
```

Rails uses a class method called `has_many` to create this association in a readable way. Now, edit *app/model/comment.rb*, and you'll see that the Rails generator already added the matching `belongs_to` statement for you automatically:

```
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

The post to comments association should now work as intended. If your Rails console was still running while you made these changes, you'll need to restart it to see the effects.

Using Associations

When you create an association in a model, Rails automatically defines several methods for that model. Use these methods, and you won't have to worry about keeping the `post_id` updated. They maintain this relationship for you automatically.

The has_many Methods

The `has_many :comments` statement you saw inside `Post` defines several methods:

`comments` Returns an Active Record relation representing the array of comments for this post

`comments<` Adds an existing comment to this post

`comments=` Replaces the existing array of comments for this post with a given array

`comment_ids` Returns an array of the comment ids associated with this post

`comment_ids=` Replaces the existing array of comments for this post with the comments corresponding to the given array of ids

Because the `comments` method returns a relation, it is commonly used with other methods. For example, you can create new comments associated with a post with `post.comments.build`, which builds a new comment belonging to this post, or `post.comments.create`, which creates a new comment belonging to this post and saves it to the database. Each of these

methods automatically assigns the `post_id` of the newly created comment. This example creates a new comment associated with your first post. You should see the new comment in the output from `post.comments`:

```
2.1.0 :028 > post = Post.first
 => #<Post id: 1, title: "First Post", ...>
2.1.0 :029 > post.comments.create :author => "Tony", :body => "Test comment"
 => #<Comment id: 1, author: "Tony", ...>
2.1.0 :030 > post.comments
 => #<ActiveRecord::Relation [#<Comment id: 1, author: "Tony", ...>]>
```

If you want to check if any comments are associated with a post, use `comments.empty?`, which returns `true` if there are none. You might also find it helpful to know how many comments are associated with a particular post; in that case, you use `comments.size`:

```
2.1.0 :031 > post.comments.empty?
 => false
2.1.0 :032 > post.comments.size
 => 1
```

When you know a post has comments associated with it, you can look for a particular comment by passing `post.comments.find` a comment id. This method raises an `ActiveRecord::RecordNotFound` exception if a matching comment cannot be found belonging to this post. Use `post.comments.where` instead if you would rather not raise an exception. This method just returns an empty relation if a matching comment is not found.

The belongs_to Methods

The `belongs_to :post` statement inside the `Comment` model defines five methods. Because `belongs_to` is a singular association (a comment can only belong to one post), all of these methods have singular names:

`post` Returns an instance of the post that this comment belongs to

`post=` Assigns this comment to a different post

`build_post` Builds a new post for this comment

`create_post` Creates a new post for this comment and saves it to the database

`create_post!` Creates a new post for this comment but raises `ActiveRecord::RecordInvalid` if the post is not valid

These methods are the inverse of the methods defined in the `Post` model. Use them when you have a comment and you would like to manipulate its post. For example, let's fetch the post associated with our first comment:

```
2.1.0 :033 > comment = Comment.first
 => #<Comment id: 1, author: "Tony", ...>
2.1.0 :034 > comment.post
 => #<Post id: 1, title: "First Post", ...>
```

Calling `post` on the first comment, which is also our only comment so far, should return our first post. This confirms the association works both ways. Assuming you still have more than one post in your database, you can also assign this comment to a different post:

```
2.1.0 :035 > comment.post = Post.last
 => #<Post id: 2, title: "Second Post", ...>
2.1.0 :036 > comment.save
 => true
```

Assigning a comment to another post updates the comment's `post_id`, but does not write that to the database. Don't forget to call `save` after updating the `post_id`! If you make this common mistake, the comment's `post_id` won't actually change.

Summary

This chapter has been a whirlwind tour of Active Record, so play around in the console until you're comfortable with these ideas. Add more posts, update the existing posts with body text, and create comments associated with these posts. Focus on the CRUD operations and association methods in particular. These methods are commonly used in all Rails applications.

The next chapter covers Rails controllers. There, you'll see all of these methods in use as you work your way through the various controller actions.

Exercises

1. It might be nice to contact the people leaving comments on our blog. Generate a new migration to add a string column to the comments table to store an email address. Run this migration, and use the Rails console to verify that you can add an email address to comments now.
2. We need to ensure that users actually enter some text when they create a comment. Add validations to the comments model for the `author` and `body` fields.
3. Write a query to determine the number of comments belonging to each post. You can't do this with a single query, but you should be able to find the answer by iterating over a collection of posts as if it were an array.