Rails and Active Record

Start Assignment

Due No Due Date **Points** 1 **Submitting** a website url

FORK (https://github.com/learn-co-curriculum/phase-4-rails-and-active-record-readme/fork)
(https://github.com/learn-co-curriculum/phase-4-rails-and-active-record-readme)
(https://github.com/learn-co-curriculum/phase-4-rails-and-active-record-readme/issues/new)

Learning Goals

- Use Rails to generate a model
- Create seed data
- Test code using the Rails console

Introduction

In this code-along, we'll discuss how Active Record fits into a Rails application, as well as some tools to make it easier to set up models in your Rails apps.

Fork and clone this repo, then run `bundle install` to download the required dependencies before coding along.

> **Note:** This code-along has been set up as a Rails app for you, so you will be running all the terminal commands inside the lesson directory.

Active Record's Role

Active Record is the built-in ORM that Rails utilizes to manage the model aspects of an application. What is an ORM? An ORM is an Object Relational Mapper system — a module that enables your application to manage data in a method-driven structure. This means that you are able to run queries, add records, and perform all of the traditional database processes by leveraging methods as opposed to writing SQL manually. For example, below is the traditional way that we would query a database of 'cheeses' using SQL:

```sql
SELECT * FROM cheeses
```

Compared with leveraging Active Record:

```ruby
Cheese.all
```

By using Active Record, you are also able to perform advanced query tasks, such as method chaining and scoping, which typically require less code and make for a more readable query.

Active Record Models

By using model files, we are able to create an organized layer of abstraction for our data. An important thing to remember is that at the end of the day the model file is a Ruby class. It will typically inherit from the `ActiveRecord::Base` class, which means that it has access to a number of methods that assist in working with the database. However, you can treat it like a regular Ruby class, allowing you to create methods, data attributes, and everything else that you would want to do in a class file.

A typical model file will contain code such as but not limited to the following:

- **Custom scopes (http://api.rubyonrails.org/classes/ActiveRecord/Scoping/Named/ClassMethods.html)**
- Model instance methods
- Default settings for database columns
- **Validations (http://api.rubyonrails.org/classes/ActiveModel/Validations/ClassMethods.html)**
- **Model-to-model relationships (http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html)**
- **Callbacks (http://api.rubyonrails.org/classes/ActiveRecord/Callbacks.html)**
- Custom algorithms

If any/all of the items above aren't familiar to you yet, don't worry. We'll cover them in future lessons. It's important to have an idea of what can be included in a model file, even at this early stage.

Creating an Active Record Model

As a Rails developer, one common task you'll have is creating models that are connected to a database via Active Record. To simplify this task, and ensure that your code follows Rails conventions, Rails has a built-in generator to help write some of the code you'll need for a new model.

Let's start with a definition of what our model will be. For our Cheese Shop application, we'll want a way to keep track of all of our cheeses. In particular, we'll want to know our cheese's *name*, its *price* in dollars, and whether it's a *best seller*. In the database, our `cheeses` should look something like this:

`id`	`name`	`price`	`is_best_seller`
1	Cheddar	3	true
2	Pepper Jack	4	true
3	Limburger	8	false

To set this up in our Rails app, we'll need a few things:

- A **migration** with instructions on how the table should be created in the database
- A **model** so we can interact with that table via Active Record

Using Rails Generators

Rails gives us an easy way of creating both of these things using a **generator** we can run from the command line.

Make sure you've navigated into the directory for this lesson, then run this command in your terminal:

```
$ rails g model Cheese name price:integer is_best_seller:boolean --no-test-framewo
```

We're telling Rails to generate the code for a Cheese model, with a `name` attribute (string, the default data type), `price` (integer), and `is_best_seller` (boolean).

IMPORTANT: Note that we have added the `no-test-framework` argument to our `rails g` command. **You should add this argument to every Rails generator command you run while you're working in the Flatiron curriculum.** Without this flag, the Rails generator will automatically create spec files that can interfere with the lab tests. If you forget to add it, you will have to comb through your local repo and manually delete all of the newly-created, unnecessary tests. This is not fun!

Please, remember the `--no-test-framework` argument.

You'll see this code added in a `create_cheeses.rb` file — the name of the file will be prepended with a time stamp — inside the `db/migrate` folder:

```
class CreateCheeses < ActiveRecord::Migration[6.1]
  def change
    create_table :cheeses do |t|
      t.string :name
      t.integer :price
      t.boolean :is_best_seller

      t.timestamps
    end
  end
end
```

As well as this code in the `app/models` folder:

```
# app/models/cheese.rb
class Cheese < ApplicationRecord
end
```

Working With Migrations

Now that we've created this migration, we have to run the migration, which will create this table for us and generate a schema file.

Run the migration by running:

```
$ rails db:migrate
```

For a refresher on migrations, see [this documentation (http://edgeguides.rubyonrails.org/active_record_migrations.html)](http://edgeguides.rubyonrails.org/active_record_migrations.html). This migration follows the standard naming convention. When you want to create a table, the migration's class name should reflect that; hence, `CreateCheeses` . This is then reiterated by the `:cheeses` argument passed to the `create_table` method. The filename itself needs to be unique, and you will notice that the migration file name is prepended with a timestamp value to make sure that we can run migrations in the order they were written.

The timestamp also plays a role in making sure that only new migrations run when we run `rails db:migrate` . The `db/schema.rb` file is updated with a version number corresponding to the timestamp of the last migration you ran. When you run `rails db:migrate` again, only migrations whose timestamps are greater than the schema's version number will run. So, the numbers at the beginning of the filenames of your migrations are required so Active Record can be sure to run each of your migrations just once and in the proper order.

After running `rails db:migrate` we can see that our `db/schema.rb` file has been updated with our new `cheeses` table!

Seeding Data

To add some initial data in our database, we can take advantage of the 'seeds' feature of Rails. We can write some code in a `db/seeds.rb` file that is responsible for setting up our database:

```
# db/seeds.rb
Cheese.create!(name: 'Cheddar', price: 3, is_best_seller: true)
Cheese.create!(name: 'Pepper Jack', price: 4, is_best_seller: true)
Cheese.create!(name: 'Limburger', price: 8, is_best_seller: false)
```

Next, run `rails db:seed` to execute all the code in the `db/seeds.rb` file.

Let's test out our code using the Rails console. Open up the console by running `rails console` or `rails c` . Running the console will load the entire Rails environment and give you command line access to the app and the database. The console is a powerful tool that you can leverage in order to test out scripts, methods, and database queries.

Once the session has started, run the following command to ensure it recognizes our new Cheese model:

```
Cheese.all
```

If everything is set up properly, you will see that it returns all the cheeses we created in the seed file! You can test out some other methods here as well:

```
Cheese.last
```

Recall that Active Record also allows us to use method chaining to access the attributes we've defined for our Cheese model:

```
Cheese.last.name
```

With our `Cheese` model working, let's add a new feature that returns a summary of a cheese. Exit out of the Rails console with `control + d`. Then add this to the Cheese model file:

```
# app/models/cheese.rb

def summary
  "#{self.name}: $#{self.price}"
end
```

Go ahead and test this new method out in the Rails console by running a query on one of our records, for example:

```
Cheese.last.summary
```

> If you didn't exit the Rails console before adding the new Cheese#summary method, you can get the console to reload all your code by running `reload!`.

It should return the summary value of the last cheese we created: `"Limburger: $8"`.

As you may have noticed, we did not have to create a controller, route, view, etc. in order to get the `Cheese` model working. The data aspect of the application can work separately from the view and data flow logic. This level of abstraction makes it possible to test the behavior of our models without having them strongly coupled to how they are rendered to the user.

Conclusion

We covered quite a bit of material in this lesson. You should now have a firm understanding of Active Record models in Rails. Active Record is a powerful tool that enables developers to focus on the logic of their applications while streamlining processes such as connecting to the database, running queries, and much more.

Check For Understanding

Before you move on, make sure you can answer the following questions:

1. What are some of the ways that using Active Record in Rails makes it easier to interact with model data stored in a database?
2. What are the advantages of using a Rails generator to create our models?
3. Why is it important for migration file names to have a timestamp prepended?