Module 5
Advanced Rails

CS W169A: Software Engineering

John Yang | Summer 2020

1 Overview

Validation and filters are two Rails features that abide by the principles of aspect oriented programming. Validations are applied to models, and are use to check certain conditions before allowing a model to save data to the database. Filters, on the other hand, are used to check certain conditions before allowing a controller action to run. In this discussion, we will explore implementing both of these in a real code base.

In the second half, we will discuss associations, which are powerful feature of Rails allowing us to define relationships between models. Rails abstracts away many of the traditional complications that come with executing a variety of database operations, such as joins, making life a lot easier for us, the programmer!

2 Validations

Say we are given the `User` model as follows:

```
class User < ActiveRecord::Base
    validates :username, :presence => true
    validate :username_format
end
```

If you need a refresher, the documentation on validations are linked here. The first `validates` acts on the `username` field. A `User` object will not be valid without a `username` attribute.

The `validate` keyword works differently from `validates`. `validate` takes a method/block (in this case, textttusername_format), and uses it to validate records when they are modified or inserted into the database. Documentation link.

1. What happens if we have `@user` with no username and we call `@user.valid?`. What will `@user.save` do? What will `@user.save!` do?
   `@user.valid?` returns false, `@user.save` returns false and won't save to the database, `@user.save!` will thrown an exception and won't save to the database.
2. Implement `username_format`. For our purposes, an username starts with a letter and is at most 10 characters long. Remember, custom validations add a message to the errors collection.
   Solution:

```
def username_format
    if username.length < 10 or not username =~ /^[a-z]/i
        errors.add(:username, "is not formatted correctly")
    end
end
```

3 Filters

Remember, filters help us check whether certain conditions hold before allowing a controller action to run. For the `User` model, let's say we want to check if `@user` was an admin for all the methods in the `AdminController`.

Fill in the `before_filter:check_admin` method below that checks if the admin field on `@user` is true. If not, redirect to the `admin_login` page with a message indicated restricted access.

Solution:

```
class AdminController < ApplicationController
    before_filter :check_admin
    def check_admin
        if not @user.admin
            flash[:notice] = "You must be an admin"
            redirect_to '/admin_login'
        end
    end
end
```

4 Associations

4.1 Setting Up Associations

For each group of models, describe what association you would add to each model and what migrations you would need to run to make the methods work.

1. `@farmer.cows`
   Farmer has_many cows, need foreign key on cow

2. `@pokemon.trainer` and `@trainer.pokemons`
   Pokemon belongs_to trainer, Trainer has_many pokemon, key on pokemon

3. `@student.majors, @major.students, @student.degrees, @major.degrees, @degree.major, @degree.student`
   Students has_many majors through degree, has_many degrees
   Major has_many student through degree, has_many degrees
   Degree belongs_to major, student, has foreign key

4.2 Life Without Associations

We want to model a one to many relationship between `User` and `Picture`; i.e. a user can own many pictures, and a picture has one owner. To do this, we added a foreign key for users onto pictures (so pictures have a field `user_id`).

How would we implement the following actions WITHOUT having `belongs_to` and `has_many` on our models?

1. Create a new Picture that belongs to `@user`
   Solution:

   ```
   Picture.create(user\_id: @user.id)
   ```

2. Delete `@user` and all of of the pictures associated with that user.
   Solution:

   ```
   @pictures = Picture.where(user\_id: @user.id)
   @pictures.each do |picture|
       picture.destroy
   end
   @user.destroy
   ```

Now, say we added `belongs_to` and `has_many` to their respective models. How would we implement the two actions above?

Solution:

```
@user.pictures.create
@user.pictures.destroy_all
@user.destroy # (better is to add dependent: destroy)
```

5 Further Reading

If you're interested in seeing associations, validations, and filters in action, check out the Community application, created by Sherman Leung, who presented the app during a CS 169 discussion as a guest lecture a couple years before. The application is meant to help a group split food and utility costs. (Disclaimer: This codebase is a couple years old, and therefore, deprecated. However, most of its core functinoality remains usable). Notice:

1. **Filters**: In the `app/controllers/application_controller.rb` file, filters are used to validate a variety of inputs before being inserted into the database.
2. **Validations**: The `apps/models/diner.rb` file uses the `validates_presence_of :name` function to verify that a `name` parameter is included. The `apps/models/group.rb` has many validations used to verify business logic of a diner.
3. **Associations**: See if you can identify where these are! Any file within the `app/models/` folder has examples of associations.