Introduction to Active Record

Topics Covered: Searching, associations, querying, validations.

In this task we will continue the introduction of the Ruby on Rails web framework by expanding your existing e-commerce application.

If you have difficulties please refer to the material on https://learn.shefcompsci.org.uk, the Rails guides at http://guides.rubyonrails.org, or ask one of the epiGenesys staff.

Please Note:
This class will continue from the previous one, so all the following exercises need to be completed on your own ecommerce app. If you did not complete **up to and including task 7** in the previous class, please clone the solution from the following repository:
`git@git.shefcompsci.org.uk:com3420-2017-18/materials/ecommerce-solution.git`
and continue using that repository. If you use the solution repository, you will need to run:
`bin/setup_labclass`
in order to setup the project.

Task 1 - Searching products

It would be useful if customers could search for specific products by name, so that they do not have to scroll through a long list of products to find what they want.

1. We need to add a new feature to the application so that customers can enter a name in a text box and click 'Search', and then only products that match that name are displayed in a table. First we need to define a new route.

   Open the routes file **`config/routes.rb`** and change:

   ```
   resources :products
   ```

   to:

   ```
   resources :products do
     post :search, on: :collection
   end
   ```

This defines a POST (i.e. submit some fields filled in on a form) path called search on products. The **`on: :collection`** code creates a path for all products and not an individual product.

If you run:

```
bundle exec rails routes
```

You can see the following path has been added:

```
search_products POST   /products/search(.:format)  products#search
```

2. We now need to create a search form in a **View**. We will add this to the top of the existing products index page.

   Open the **`app/views/products/index.html.haml`** file and add the following code **in bold** after the product count text we added in the previous session:

```
%p There are #{@products.size} products available.

= simple_form_for :search, url: search_products_path, method: :post do |f|
  = f.input :name

  = f.submit 'Search'
  = link_to 'Reset', products_path

%br
```

The first argument to the **`simple_form_for`** method is the name of the parameters hash we can access in the controller when we submit the search form. For example, if we enter 'Laptop' in the name field in the form above and click the Search button here is how the form parameters are passed to the controller:

```
{"search" => {"name" => "Laptop"}}

# This can be accessed in the controller using:

params[:search][:name]
```

The **`url`** option states which path to submit the form parameters to when the Search button is clicked, in this case the search products path we created earlier. The **`method`** option states which HTTP verb to use and it must match what is specified for the url path in the routes file, in this case **post**.

The block passed to **`simple_form_for`** - indicated by **`|f|`** - enables us to add form inputs, labels, buttons, etc. The GitHub page explains all of the possible options in more detail: https://github.com/plataformatec/simple_form

3. We have created the path and view successfully, but now we need to add a **Controller** method so that when we **POST** some fields from our view to the search action it does something with them. The **`search_products`** path above states that the controller and action combination needs to be **`products#search`** so we need to add a **`search`** action in our **`products`** controller.

   Open the **`app/controllers/products_controller.rb`** file and add the following code **in bold** after the **`destroy`** action:

```
# DELETE /products/1
def destroy
  @product.destroy
  redirect_to products_url, notice: 'Product was successfully
destroyed.'
end

# POST /products/search
def search
  # {"search" => {"name" => "some entered name"} }
  @products = Product.where(name: params[:search][:name])
  render :index
end
```

   We can access what was entered in the name field on the search form using the parameters hash. It is a hash of hashes which is why to access the name entered we use the following code:

```
params[:search][:name]
```

   To find the correct product(s) we need to query the database. We could write SQL (Structured Query Language) ourselves. However, Rails helpfully provides us with simple class methods on models that enable us to pass in parameters and let Rails handle the querying and fetching of records.

   One of these methods is **`where`**, which takes a hash of options and retrieves all records that match them in the database. For example, **`Product.where(name: params[:search][:name])`** retrieves all product records where the name equals what was entered in the search form.

   The result of the **`where`** method is an array of objects, and this is assigned to the instance variable **`@products`**. The final line **`render :index`** renders the **`view`** for the **`index`** action, which is **`app/views/products/index.html.haml`**, and passes the **`@products`** instance variable to it.

4. At this point, we have enough code that we should commit what we have done to Git.

Task 2 - Associating categories to products

1. We can currently manage categories, which have a code and a name, but we have no way of associating them with products. We will do this now.

   Create a standalone migration that adds a **`category_id`** integer column to the products table. Remember to run the migration after doing this:

   ```
   bundle exec rails db:migrate
   ```

   See http://guides.rubyonrails.org/active_record_migrations.html#creating-a-migration for information on how to do this. This will create a foreign key which allows you to associate a product to a category using the id in the database.

2. Add a **`belongs_to`** association so that a product **`belongs_to`** a category. See http://guides.rubyonrails.org/association_basics.html#the-belongs-to-association for information.

3. Add a link to the menu at the top of the application to access the categories index page, if you have not already done so.

4. Add a new field to **`app/views/products/_form.html.haml`** that allows you to assign a category when creating a product - use the field identifier 'category_id'. You might want to use a **select box** to do this - see https://github.com/plataformatec/simple_form#associations for information.

5. You will need to permit 'category_id' in the products controller. You do this in the method **`product_params`**.

6. Display the category on the products index page, and product show page.

Task 3 - Validations

A very common task when building web applications is adding validations that check user input to ensure it is present and sensible. Active Record provides us with lots of tools to make this easy.

Validations for **Product** could be added to **`app/models/product.rb`** like this:

```
validates :name, :description, :cost, :category_id, presence: true
validates :cost, numericality: { greater_than: 0 }
```

The above tells Rails that the name, description and cost all have to be present, and that the cost must be greater than 0. Validating the presence of **category_id** forces the user to choose a category in order to create a product.

1. Implement some sensible validations for your **Category** model.

Task 4 - Scopes

Scopes are a feature of Active Record that enable us to group a collection of records. This is typically a subset of one type of model that all share a common property or properties. We will now implement a feature to deactivate old categories.

1. Generate a migration to add a **boolean** field to the **categories** table, and call this field **active**.

   Given that we probably have existing records in the database it is a good idea to give this field a sensible default. In this case we will set it to true.

   ```
   add_column :categories, :active, :boolean, default: true
   ```

2. Add the new **active** field to the categories form. Do not forget to allow it as a parameter in the controller.

3. Write a scope to return only the active categories. We do this in the **category** model.

   ```
   scope :active, -> { where(active: true) }
   ```

   The above defines a scope for categories which will return only the categories with the active field set to true.

4. You can check that the above is working by editing a few categories to make them inactive and then changing the **categories** controller to use this scope.

   You will have something that looks like:

   ```
   def index
     @categories = Category.all
   end
   ```

   Which should be changed to use the scope:

   ```
   def index
     @categories = Category.active
   end
   ```

Task 5 - Querying with associations

We will now amend the product search feature from Task 1 to also allow customers to search for products by category.

1. Add a select box to your existing products search form:

```
= f.input :category_id, as: :select, collection: Category.active
```

   - **`as: :select`** - tells Simple Form that you would like a select box, rather than a text box.
   - **`collection: Category.active`** - tells Simple Form to only include active categories using the scope we added in Task 4.

2. The only other thing needed is to handle this in the products controller:

```
def search
  @products = Product.where(category_id: params[:search][:category_id])
  @products = @products.where(name: params[:search][:name]) if
params[:search][:name].present?

  render :index
end
```

   There are two things to notice here. Firstly, we have added the line:

```
@products = Product.where(category_id: params[:search][:category_id])
```

   Similarly to before, we have obtained the **`category_id`** from the form parameters and retrieved products associated with that category.

   Secondly, we have this line:

```
@products = @products.where(name: params[:search][:name]) if params[:search][:name].present?
```

   At first glance this may look like we are overwriting **`@products`**, but what is happening here is that Active Record is chaining queries together.

   `@products` will initially represent all of the products associated with the specified category, but the query will then be amended with an additional constraint of having a name that has matched the search (but only if a name was searched for).

Task 6 - Many-to-many associations

Many-to-many relationships can be defined with a **has_and_belongs_to_many** association. This is a simple many-to-many relationship implemented using a very simple linker table.

You can read more about these at:
http://guides.rubyonrails.org/association_basics.html#the-has-and-belongs-to-many-association

We are going to simulate this within our ecommerce application by adding tags to categories.

1. To begin, create a new 'tag' model:

```
bundle exec rails g model Tag name:string
```

2. Migrate your database to add the new table to it.

3. Generate a new migration to create the linker table:

```
bundle exec rails g migration create_join_table_categories_tags categories:index tags:index
```

4. Open the file that is generated, and it should look something like this:

```
class CreateJoinTableCategoriesTags < ActiveRecord::Migration[5.1]
  def change
    create_join_table :categories, :tags do |t|
      t.index [:category_id, :tag_id]
      t.index [:tag_id, :category_id]
    end
  end
end
```

   If everything looks correct, migrate your database as before to generate the linker table.

5. You can generate some example tags by running the following command:

```
bundle exec rails runner '(1..8).each { |num| Tag.new(name: "Tag #{num}").save! }'
```

   If you see an error at this point, make sure you have followed the above steps correctly, and that you have migrated your database to add the two new tables above.

6. Before you can start associating tags with categories, you need to tell ActiveRecord about the relationship between the two models.

   Update the category model first so that the line **in bold** below is added to it:

```
# app/models/category.rb
class Category < ApplicationRecord
  has_and_belongs_to_many :tags
  # ...
```

   Then update the tag model with the line **in bold** below:

```
# app/models/tag.rb
class Tag < ApplicationRecord
  has_and_belongs_to_many :categories
  # ...
```

7. Edit the view related to your categories:

   a. If you have used the solution repository, this is in **app/views/categories/_form.html.haml**

   b. If you have continued working on your existing ecommerce application, this may be inside **app/views/categories/new.html.haml**

8. Add the following line inside the view to enable the addition of tags to a category:

```
= f.association :tags, input_html: { class: 'select2' }
```

9. The final thing you need to do is update the controller to permit the setting of tags when you create or update your categories:

```
...
  def category_params
    params.require(:category).permit(:name, :code, tag_ids: [])
  end
...
```