Building a Rails Application (Part Two)

Topics Covered: jQuery, Ajax, Uploading, Importing

Getting started with this worksheet

- Before you can begin, you must have uploaded an SSH key to GitLab using instructions in the *Using Ruby on Rails with CiCS Linux* document, and then *Creating and Deploying Your Team Project* as a team.
- If you have difficulties please refer to the videos on https://learn.shefcompsci.org.uk, the Rails guides at http://guides.rubyonrails.org, or ask one of the demonstrators.

Please Note:
This class requires a specific sample application, as it contains seed data required to complete the tasks within this worksheet. Please clone the sample from the following repository:
`git@git.shefcompsci.org.uk:com3420-2017-18/materials/week-3a-sample.git`
Change into the **week-3a-sample** directory and run the following command:
`bin/setup_labclass`
in order to setup the project.

Introduction to jQuery

jQuery is a JavaScript library that wraps a number of common JavaScript tasks (for example, traversal of the DOM tree of your website or event handling) within a simple API. The library is generally cross-browser compatible, so the JavaScript code does not need to be customised to handle different browser engines.

Our Rails template bundles jQuery support within the application, so there is no need to include a separate jQuery library if you want to make use of the most common functions. You can read more about jQuery at https://jquery.com/

Task 1 - Implement quick search using jQuery

The sample application provided contains a number of example products that are added when you initially set it up.

1. Open the **app/views/products/index.html.haml** file within your editor.
2. Above the existing search form, add a new link as follows:

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

%ul
  %li
    = link_to '#', id: 'search-books' do
      Quick search for Books
```

3. If you open the page in your browser and click the 'Quick search for Books' link, you'll notice that nothing happens.
4. Create a new JavaScript file within **app/assets/javascripts** called **quick_search.js**.
5. Within the new JavaScript file, add the following code:

```
$(document).ready(function(){
  $('#search-books').click(function(){
    $('#search-category option:contains("Books")').attr('selected', 'selected');
    $('#search-button').click();
  });
});
```

   a. The **$(document).ready(...)** line tells jQuery to wait until the page is loaded before running the code inside it.
   b. The **$('#search-books').click(...)** line is an event handler - it tells the browser that when the relevant link is clicked, call the JavaScript code inside it.
   c. The **$('#search-category option:contains(...)').attr(...)** line searches the dropdown for an option that contains the text 'Books' and then marks it as the selected option.
   d. The **$('search-button').click()** line tells jQuery to call the click action on the search button - which submits the search form and refreshes the screen.

6. Add the **quick_search.js** file to the JavaScript manifest file located in **app/assets/javascripts/application.js.**
7. Refresh the page, and click the 'Quick search for Books' link to check it is working.
8. Using the knowledge you've gained from above, add some additional quick search links, based on the text below:
   a. Quick search for Harry Potter
   b. Quick search for Pink Floyd in Media

Introduction to Ajax

Ajax is a way of sending and receiving data without refreshing or reloading the current page. You can use jQuery to make an Ajax request directly (http://api.jquery.com/jquery.ajax/), but this class is going to focus on how to do it via the Rails integrations that come with our template.

Task 2 - Editing products with Ajax

We will now use a combination of Rails and our epiJs (https://github.com/epigenesys/epi-js) gem to edit products via Ajax.

1. Change your view file to look like a modal dialog.

First edit your **app/views/products/edit.html.haml** file to look like this:

```
.modal-dialog
  .modal-content
    = simple_form_for @product do |f|
      .modal-header
        %button.close{ type: :button, data: { dismiss: :modal } }
          %span &times;
        %h4.modal-title Editing product
      .modal-body
        = f.input :name
        = f.input :description
        = f.input :cost
        = f.association :category

      .modal-footer
        .pull-left
          = dismiss_modal_button
        = f.button :submit
```

If you now visit the edit page, you will notice it has the appearance of a Bootstrap modal dialog.

2. Load the view via Ajax

Now your view looks like modal, you need to load it via Ajax. Open the **app/views/products/_products_table.html.haml** partial and locate the line with the 'Edit' link on it.

If you give the link a CSS class of *'ajax-modal'*, the epiJS gem will load the view via Ajax.

```
%td= link_to 'Edit', edit_product_path(product), class: 'ajax-modal'
```

If you refresh the page you will notice that the modal loads, but that the page is a bit untidy: the footer is displayed incorrectly, and the navbar has not greyed out like the rest of the

page. The reason for this is that Rails is re-rendering the default layout of the application when calling the edit page via Ajax.

To stop Rails from loading the layout, you need to edit the products controller and tell Rails not to load the layout for the edit action:

```
# GET /products/1/edit
def edit
  render layout: false
end
```

If you click the 'Edit' link again, the modal should be rendered properly.

3. Submit the form via Ajax and respond with JavaScript

Despite rendering the edit form as a modal, clicking the submit button within the modal will still cause the entire page to be refreshed - this isn't necessarily ideal as it will then re-render the page separately to the table, rather than updating using the changes in the modal dialog.

The first step to achieving this is to get the edit form to submit via Ajax. Rails has a helper built into it to allow you to do that - simply add **remote: true** to your form definition:

```
.modal-dialog
  .modal-content
    = simple_form_for @product, remote: true do |f|
      // The rest is unchanged
```

You now need to create two new views - one that handles the success condition (the validation has passed and the model has been updated) and a second that handles the failure condition (validation has failed).

Create a new view called **update_success.js.haml** in **app/views/products** and add the following code to it:

```
$('#modalWindow').modal('hide');
$("#products-table").html($('#{j render 'products_table'}'));
```

This is JavaScript code we want to be executed when we update the product successfully. It closes the model and reloads the products table.

Create another new view in the same directory called **update_failure.js.haml** and add the following to it:

```
$('#modalWindow').html('#{j render template: 'products/edit'}');
```

This is JavaScript code we want to be executed when updating the product fails - when there is a validator failure for example. It re-renders the modal, allowing the errors to be displayed.

The last thing you need to do is update the controller to render the JavaScript views you have just made. Edit the products controller and replace the update action with the following code:

```
def update
  if @product.update(product_params)
    @products = Product.all
    render 'update_success'
  else
    render 'update_failure'
  end
end
```

If you now open up an edit modal for one of your products, submitting a change will re-render the table rather than refresh the whole page.

Task 3 - Uploading files to your application

If you want users to be able to upload files such as images that need to be stored for later use, you can use the Carrierwave gem to handle these uploads and place them on the disk for you.

Carrierwave isn't included in our Rails template by default, so you need to add it yourself.

Edit the **Gemfile** (not Gemfile.lock) and add the following line below the 'pg' gem:

```
gem 'sqlite3', group: [:development, :test]
gem 'pg'
gem 'carrierwave'
```

Once you've add the gem, install it by running the following command:

```
bundle install
```

1. Create an uploader

We will use Carrierwave to upload images to categories. In order to use Carrierwave we need to generate a new uploader:

```
bundle exec rails g uploader CategoryImage
```

This will create a new folder under **app** for storing uploaders, and create a new file called **category_image_uploader.rb**.

You will need to restart your Rails server at this point to load the new uploader.

Open the new uploader (**app/uploaders/category_image_uploader.rb**) and take a look. The file contains a lot of commented out sample code. For this class you can remove most if it, but we want to keep the **extension_whitelist** method**.**

If you uncomment the extension_whitelist method and remove the additional sample code, your file should look something like this:

```
class CategoryImageUploader < CarrierWave::Uploader::Base
  # Choose what kind of storage to use for this uploader:
  storage :file

  # Override the directory where uploaded files will be stored.
  # This is a sensible default for uploaders that are meant to be mounted:
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Add a white list of extensions which are allowed to be uploaded.
  # For images you might use something like this:
  def extension_whitelist
    %w(jpg jpeg gif png)
  end
end
```

The only other thing you need to change in this file is the **store_dir** method. By default this directory is relative to the application's **public** folder. Anything stored in the public folder is readable to anybody on the internet without authorisation. It is more often the case that we wish to carefully control who can see what has been uploaded to our applications, so we need to change this to be a directory outside the public folder:

```
# ...
def store_dir
  Rails.root.join("uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}")
end
# ...
```

2. Create a new field in your database

To use the uploader you need to add a new field to your products table. This field is used by Carrierwave to save the path to the uploaded file.

Create the new migration by running the following command:

```
bundle exec rails g migration AddCategoryImageFileToProducts
```

Open the new migration file by locating it in the **db/migrate** folder and then add the following line to the **change** method:

```
add_column :categories, :category_image_file, :string
```

Once the migration file is updated, update your database by running the migration:

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

3. Mount your uploader on the Category model

Before you can start uploading images to your categories, you need to tell Carrierwave which field is being used to store the image information. You do this by 'mounting' the uploader within your model.

Add the following to your category model class:

```
class Category < ApplicationModel
  # ...
  mount_uploader :category_image_file, CategoryImageUploader
  # ...
end
```

This will create a virtual attribute on the model, **category_image_file**, which can then be included in your form as a new input.

4. Update your form to allow image uploads

Open the category form at **app/views/categories/_form.html.haml** and add the two new fields below:

```
# ...
= f.input :code
= f.input :name

= f.input :category_image_file, as: :file
= f.hidden_field :category_image_file_cache

= f.input :active, as: :inline_radio_buttons
# ...
```

The first one is the file picker, and the hidden field is needed to persist the file between page renders (if you have a validation error for example).

You also need to add the new fields to the permitted parameters in the categories controller:

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

5. Show the image in your application

Due to the changes made to the **store_dir** method earlier, the images that are uploaded to your application are stored outside of the **public** folder, which means they're not accessible directly from your application.

Firstly, you need to add a new action to the categories controller:

```
def show_image
  @category = Category.find(params[:id])
  send_file @category.category_image_file.url, disposition: 'inline'
end
```

This action will try to find the category you have requested, and then display the associated image for that category. Implicitly adding **disposition: 'inline'** tells the browser you'd prefer the attached file was displayed in the browser, rather than downloaded onto the user's machine.

You will now need to edit the routes file to make the action accessible. Update your **config/routes.rb** and edit the categories resource to match the changes below:

```
resources :categories do
  get :show_image, on: :member
end
```

Finally, you want to display the category image on the show page for that category. Edit the **show.html.haml** file inside the categories view folder and add the following line after the category code:

```
= image_tag show_image_category_path(@category)
```

Visit the show page for a category where you have uploaded an image, and the associated image should appear on the page.

Task 4 - Importing data into your application

Often it is required to upload data to your application in bulk, and one of the most common ways of doing this is via an import from a spreadsheet. This example should get you started with a very basic method of importing a spreadsheet in the CSV format.

The sample application includes a new tab on the main menu - 'Uploads'. We have provided an example file that can be uploaded to the application in the **csvs** directory at the root of the project. This process doesn't use Carrierwave, as we don't need to retain the CSV file once it has been processed by the import method.

If you visit the uploads page in your browser and try to upload a file now, it will return an error message.

Open the **app/services/imports/importer.rb** file and add the following code within the import method:

```
def import
  csv = CSV.read(@path, headers: true, skip_blanks: true)
  csv_valid = (['name', 'cost', 'description'] - csv.headers.compact).empty?
  return false unless csv_valid

  csv.each do |product|
    new_product = Product.new
    new_product.name = product["name"]
    new_product.cost = product["cost"]
    new_product.description = product["description"]
    new_product.save!
  end

  return true
end
```

The code above reads the CSV file given, and validates that three headers - name, cost and description - exist within the file.

If the file is valid, a new product is constructed from the data given in the CSV file. If the product fails to save, an exception will be displayed.

There is a sample CSV file provided in the **csvs** folder within the project. Feel free to add additional rows and see what happens when you import the file.

This is a very basic example to get you started. If you need customer facing import functionally in your project, you need to think about better error handling and reporting.