COMM049: Mobile Application Development

Improving the Experience Autumn 2016

Tidying Everything Up

By the end of the last lab class you were able to store new articles in the database and associate comments with them. This was all done with minimal coding by generating scaffolds for both the Article and Comment models. We needed to modify things a little to:

1) Build the association between articles and comments;
2) Automatically assign the appropriate foreign key to a new comment for a specific article.

However, things are not quite perfect, as some of you will have noticed. There is a big thing that we need to do and that is to start to style the website. But there is also a fundamental issue with the Comments. We have an index page for all comments. But this is not really what we want. It will be far more relevant to get a list of comments associated with a specific article. We will first generate this in a page of its own. Once this is sorted out we will then merge this into the "show" page for their associated article. After that, we will start on a more rigorous styling of the whole site.

Focusing on Comments for a Specific Article

What we would like to be able to do is to add a button to the page that displays the details of a specific article so that we can list all the comments associated with it. We already have a path to the index page to all comments: `comments_path`. We can create a link to this, but we want to modify this by passing a parameter which is the id of the article we are currently looking at.

We will do this by adding a key/value pair to the comments_path. Update the links at the end of `app/views/articles/show.html.erb` by adding the line in bold below:

```
<%= link_to 'Edit', edit_article_path(@event) %> |
<%= link_to 'List Comments', comments_path(article_id: @article) %> |
<%= link_to 'Back', articles_path %>
```

Remember that this uses Rails' shorthand for a hash; the key is `article_id`, and Rails implicitly takes as value the `id` of `@article` rather than pass the whole object (see the last lab sheet for the full explanation of this).

Now we need to pick this parameter up in the index action of the comments controller. This will then be used to collect only those comments whose foreign key matches the id of the article we were just showing.

So, change one line and add another in the index action of `app/controllers/comments_controller.rb` so that it looks like this (changes in bold:

```
  def index
    @comments = Comment.where(article_id: params[:article_id])
    @article = Article.find(params[:article_id])
  end
```

The `where` method in Rails is extremely useful for generating database queries. The pattern we have used here will return the union of comments where the key, article_id, equals the value of the parameter (article_id) in the http request that called the index action. I will cover more on this in the lectures.

Now we have the correct subset of the comments, it will be useful to modify `app/views/comments/index.html.erb` so that includes the title of the article whose comments are being displayed. This is why I have fetched the article and assigned it to an instance variable. We can then use a matching instance variable in the view thus:

```
<h1>Listing comments for <%= @article.title %></h1>
```

You might also want to update the link at the end of this file to:

```
<%= link_to 'New Comment', new_comment_path(article_id: @article) %>
```

There three more changes to make. You will notice that when you show, edit or create a new comment there is a “Back” button which formerly took you back to the full index of comments. That won’t work now, as the index action in the controller needs to be passed the foreign key of the comments it is to display. So, in each of:

```
app/views/comments/edit.html.erb
app/views/comments/new.html.erb
app/views/comments/show.html.erb
```

modify the link to the Back button so that it returns to the correct subset of comments:

```
<%= link_to 'Back', comments_path(article_id: @comment.article_id) %>
```

Placing Comments on the same page as their Article

The scaffold was all very nice for getting us up and running, but it has left us with a structure that is too fragmented. This is typical of how things work with Rails – you use generators to get things up and running quickly and then progressively refine at a detailed coding level to get the final product.

What we want to do now is to move the index of comments onto the same page that is showing their associated article. We don’t need to do much to make this happen though. We will turn the index page for the comments into a partial, and then render that on `app/views/articles/show.html.erb`. In the process of doing that, we will see how to pass parameters into a partial.

Turning the index page into a partial is trivial – just prepend an underscore so that you now have: `app/views/comments/_index.html.erb`

We are going to need access to the @comments instance variable when `app/views/articles/show.html.erb` is rendered. So you will need to modify the show controller action in app/controllers/articles_controller.rb as follows (change in bold):

```
  def show
    @title = "Article"
    @article = Article.find(params[:id])
    @comments = Comment.where(article_id: params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.json { render json: @article }
    end
  end
```

Now all that is left to be done is to render the partial when showing an article. The only really new thing here is that in order to have full control over the transfer of variables into the partial, we will explicitly pass them in using a hash when the partial is rendered. We just add the lines in bold to app/views/articles/show.html.erb:

```
<%= link_to 'Edit', edit_article_path(@event) %> |
<%= link_to 'New Comment', new_comment_path(article_id: @article) %> |
<%= link_to 'List Comments', comments_path(article_id: @article) %> |
<%= link_to 'Back', articles_path %>

<section>
  <%= render 'comments/index', comments: @comments, article: @article %>
</section>
```

What this is going to do is to pass in the values of `@comments` and `@article` using the locally scoped variables `comments` and `article`.

In order to pick this up correctly in the partial, we will need to make the changes in bold to `app/views/comments/_index.html.erb`:

```
<h1>Listing comments for <%= article.title %></h1>

<table>
  …
<% comments.each do |comment| %>
  …
</table>
<%= link_to 'New Comment', new_comment_path(article_id: article) %>
```

Note that in three places in the above reference to an instance variable has been replaced by reference to a local variable.

The slightly tricky thing about this is that I could have left the instance variables in the partial unchanged and it would still have worked. This is because the partial is being rendered within the context of `app/views/articles/show.html.erb`, and that view in turn can obtain copies of the values of all the instance variables in the associated `show` action of the `articles_controller`. However, it is *not* good practice to make use of this as it limits the applicability of the partial – it can only be used in a specific context. By changing to using local variables, we have made the partial genuinely reusable across the application at the minor cost of having to explicitly pass the values of the local variables into the partial when it is rendered.

<u>Exercise:</u>

You will find that some of the other links are broken now. Take a look around the executing application and use your judgement to use appropriate paths to fix broken links. You will also notice that there is some duplication of buttons on the show page. Don't worry about that for the moment as we will tidy that up in the next section.

You will have noticed that I have wrapped the comments up in a section. That is going to be quite useful, as we shall see now.

Revealing and hiding comments

I should warn you that I take a slightly different approach to some of the Rails texts with regards to dynamic html. Rails has some built in support for generating JavaScript. However, my preference is just to write what I want in pure JavaScript. I think this is cleaner and from a teaching perspective it will make what you learn applicable across frameworks.

Acually, we are going to do what most people do and use the jQuery library instead of writing native JavaScript.

One key thing is that we are going to make our JavaScript "unobtrusive". That is, we will not include any JavaScript within the html documents – you should have learnt not to put styling in the html, so we shouldn't put JavaScript in their either should we ☺.

Let's start off by adding a simple function that hides the comments when the page is first loaded. We do this by using a special function jQuery provides that executes when the DOM has been parsed.

First we need to identify the element that we are going to hide and then show. We will do this by given the relevant section in app/views/events/show.html.erb an identifier (change in bold):

```
<section id="comments">
  <%= render 'comments/index', comments: @comments, article: @article %>
</section>
```

Then, add this to the end of your app/assets/javascripts/application.js file:

```
$(function() {
  $( "#comments" ).hide();
})
```

The first `$()` wraps the function that will be invoked when the DOM has been parsed. The second `$()` locates the respective element (whose `id` is `'comments'`) and invokes the jQuery function `hide()` on it. Try it now – you should see that the comments are now hidden.

Now, let's put a button in `app/views/articles/show.html.erb` that can be used to reveal the comments:

```
<section id="comments">
  <%= render 'comments/index', comments: @comments, article: @article %>
</section>
```

```
<button id="show_comments">Show Comments</button>
```

As before, we will bind the functionality to the button in the app/assets/javascripts/application.js JavaScript file:

```
$(function() {
  $( "#comments" ).hide();
  $( "#show_comments" ).click(function() {
    $( "#comments" ).slideDown();
  });
})
```

What we are doing here is to bind a function to the `show_comments` button that is invoked when the button is clicked. That function will smoothly reveal the `comments` section. Try this now.

Next, we need to provide a button to hide the comments again. Add the lines in bold to the `app/views/comments/_index.html.erb` partial:

```
<%= link_to 'New Comment', new_comment_path(article_id: article) %>
<br>
<button id="hide_comments">Hide Comments</button>
```

We need to bind a function to the click event on this button:

```
$(function() {
  $( "#comments" ).hide();
  $( "#hide_comments" ).click(function() {
    $( "#comments" ).slideUp();
  });
  $( "#show_comments" ).click(function() {
    $( "#comments" ).slideDown();
  });
})
```

We are not quite there yet. The “show comments” button is not relevant when the comments are already being shown. Similarly, it needs to be revealed when the comments are hidden. Rather than have this button suddenly hidden and revealed, respectively, we can slow the process down by adding a parameter to the “hide” and “show” functions (one of “slow”, “normal”, “fast”):

```
$(function() {
  $( "#comments" ).hide();
  $( "#hide_comments" ).click(function() {
    $( "#comments" ).slideUp();
    $( "#show_comments").show('normal');
  });
  $( "#show_comments" ).click(function() {
    $( "#comments" ).slideDown();
    $( this ).hide('normal');
  });
})
```

Job done! The layout is not quite perfect (see next section), but you should find the functionality quite satisfying.

Add a little style:

I know some of you have been styling the application as you have been working on it. Now is the time to show off what you can do. Make sure:

- The header stand out distinctly;
- The navigation element displays as a horizontal bar under the header;
- The details of an event are shown in a distinct section to its associated comments (when the latter are revealed);
- You are *not* using "divs".

Please make sure that you are conforming to W3C guidelines.