**Lab 5 Part 5**

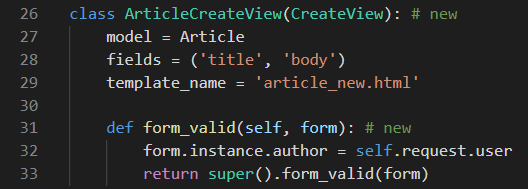
**Permissions and Authorization**

In this exercise we will add functionality to the project in order to require a user to log in to view any articles. This is known as authorization. It’s common to set different rules around who is authorized to view areas of your site. Note that this is different than authentication which is the process of registering and logging-in users. Authorization restricts access; authentication enables a user sign up and log in.

As a mature web framework, Django has built-in functionality for authorization that we can quickly use. In this exercise we will limit access to various pages only to logged-in users.

**Improved CreateView**

At present the author on a new article can be set to any user. Instead it should be automatically set to the current user. The default CreateView provides a lot of functionality for us but in order to set the current user to author we need to customize it. We will **remove** author from the fields on line 28 below and instead set it automatically via the form\_valid method.



Remove the author field on line 28

Start the server and create a new article. If you log into the Django admin you will see it is automatically set to the current logged-in user.

**Authorizations**

There are multiple issues around the lack of authorizations in our current project. Obviously we would like to restrict access to only users so we have the option of one day charging readers to read our newspaper. But beyond that, any random logged-out user who knows the correct URL can access any part of the site.

Consider what would happen if a logged-out user tried to create a new article? To try it out, click on your username in the upper right corner of the nav bar, then select “Log out” from the dropdown options. The “+ New” link disappears from the nav bar but what happens if you go to it directly: http://127.0.0.1:8000/articles/new/?

The page is still there as shown below.

Logged out new



Now try to create a new article with a title and body. Click on the “Save” button and an error message appears. This is because our model expects an author field which is linked to the current logged-in user. But since we are not logged in, there is no author, and therefore the submission fails.

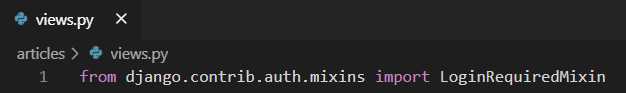


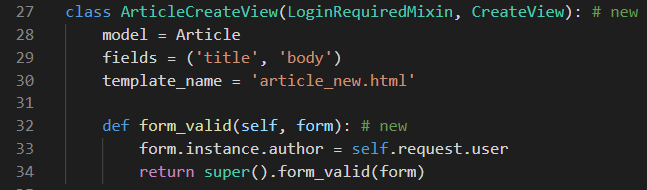
**Mixins**

We need to set some authorizations so only logged-in users can access the site. To do this we can use a mixin, which is a special kind of multiple inheritance that Django uses to avoid duplicate code and still allows customization. For example, the built-in generic ListView needs a way to return a template. But so does DetailView and in fact almost every other view. Rather than repeat the same code in each big generic view, Django breaks out this functionality into a “mixin” known as TemplateResponseMixin.

Both ListView and DetailView use this mixin to render the proper template. To restrict view access to only logged in users, Django has a LoginRequired mixin that we can use. It’s powerful and extremely concise.

In the articles/views.py file import it at the top and then add LoginRequiredMixin to our ArticleCreateView. Make sure that the mixin is to the left of CreateView so it will be read first. We want the CreateView to already know we intend to restrict access





Now return to the homepage briefly at http://127.0.0.1:8000/ so we avoid resubmitting the form. Try the link for creating new messages again: <http://127.0.0.1:8000/articles/new/>. It now redirects users to the log in page.

Now we see that restricting view access is just a matter of adding LoginRequiredMixin at the beginning of all existing views. Let’s update the rest of our articles views since we don’t want a user to be able to create, read, update, or delete a message if they aren’t logged in.



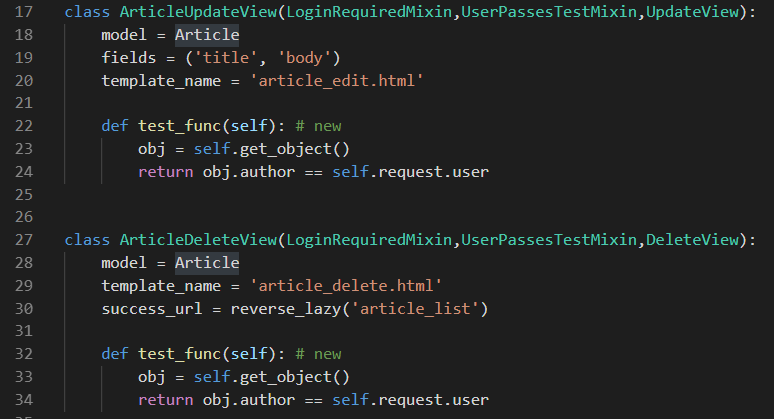
Go ahead and play around with the site to confirm that the log in redirects now work as expected. If you need help recalling what the proper URLs are, log in first and write down the URLs for each of the routes for create, edit, delete, and all articles.

We would like to further restrict the editing and deletion of articles. At the moment any logged in user can make changes to any article. What we want is to restrict this access so that only the author of an article has this permission.

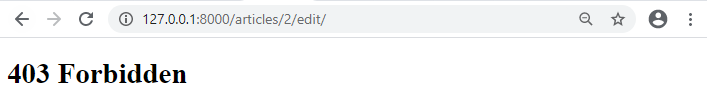
We could add permissions logic to each view for this but a more elegant solution is to create a dedicated mixin, a class with a particular feature that we want to reuse in our Django code. We can use another built-in mixin, UserPassesTestMixin, for this purpose. To use UserPassesTestMixin we first import it at the top of the articles/views.py file and then add it to both views where we want this restriction: ArticleUpdateView and ArticleDeleteView. See the code highlighted below.

The test\_func method is used by UserPassesTestMixin for our logic. We need to override it. In this case we set the variable obj to the current object returned by the view using get\_object(). Then we say, if the author on the current object matches the current user on the webpage (whoever is logged in and trying to make the change), then allow it. If false, an error will automatically be thrown.





Now log out of your superuser account and log in with testuser. If the code works, then you should not be able to edit or delete any posts written by your superuser, which is all of them right now. Instead you will see a Permission Denied 403 error page.



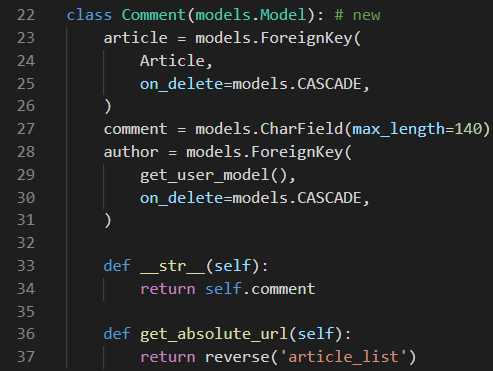
**Adding a Comments section to the web site**

In order to add “Comments” functionality to our website we will add an additional model called Comment to our articles app and link it to the Article model through a foreign key.

**Model**

To start we can add another table to our existing database called Comment. This model will have a many-to-one foreign key relationship to Article: one article can have many comments, but not the other way around. Traditionally the name of the foreign key field is simply the model it links to, so this field will be called article. The other two fields will be comment and author.

Open up the file articles/models.py and underneath the existing code add the following.



Since we’ve updated our models it’s time to make a new migration file and then apply it. Note that by adding articles at the end of the makemigrations command–which is optional–we are specifying we want to use just the articles app here. This is a good habit to use. For example, what if we made changes to models in two different apps?

If we did not specify an app, then both apps’ changes would be incorporated in the same migrations file which makes it harder, in the future, to debug errors. Keep each migration as small and contained as possible.

**Command Line**

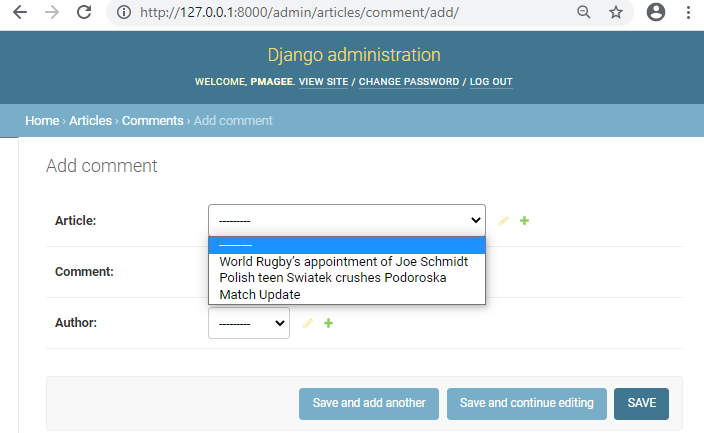
(env) djangoprojects\lab5>python manage.py makemigrations articles

(env) djangoprojects\lab5> python manage.py migrate

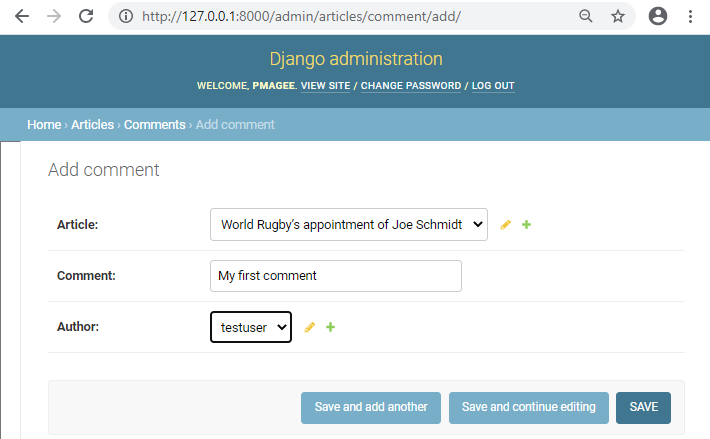
**Admin**

Add the following line of code to admin.py so that the Comment model is visible in Django admin. Log into Django Admin and you will now see two tables, Comments and Articles.

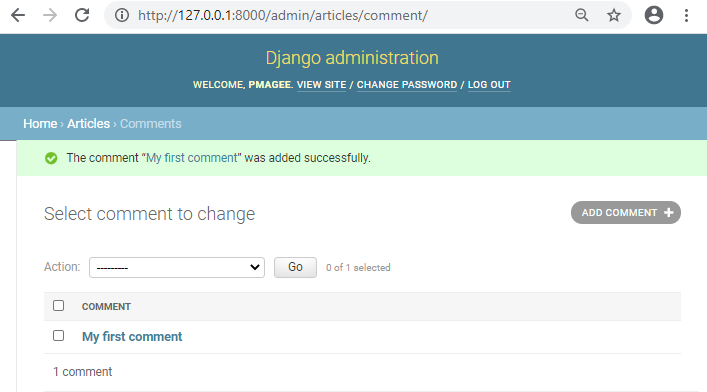
Click on the “+ Add” next to Comments. You’ll see that under Article is a dropdown of existing articles, same thing for Author, and there is a text field next to Comment.



Select an Article, write a comment, and then select an author that is not your superuser, perhaps testuser as I’ve done in the picture. Then click on the “Save” button.



You should see your comment on the Comments page.



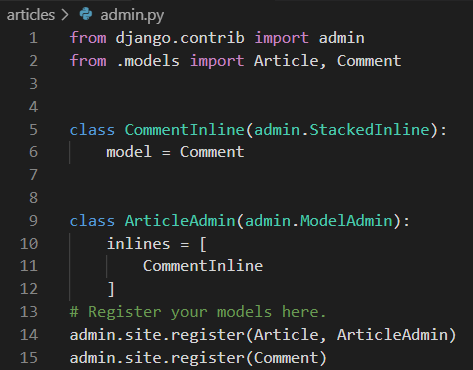
In order to see all Comment models related to a single Post model we can use a Django admin feature called inlines which displays foreign key relationships in a nice, visual way.

There are two main inline views used: **TabularInline** and **StackedInline**. The only difference between the two is the template for displaying information. In a TabularInline

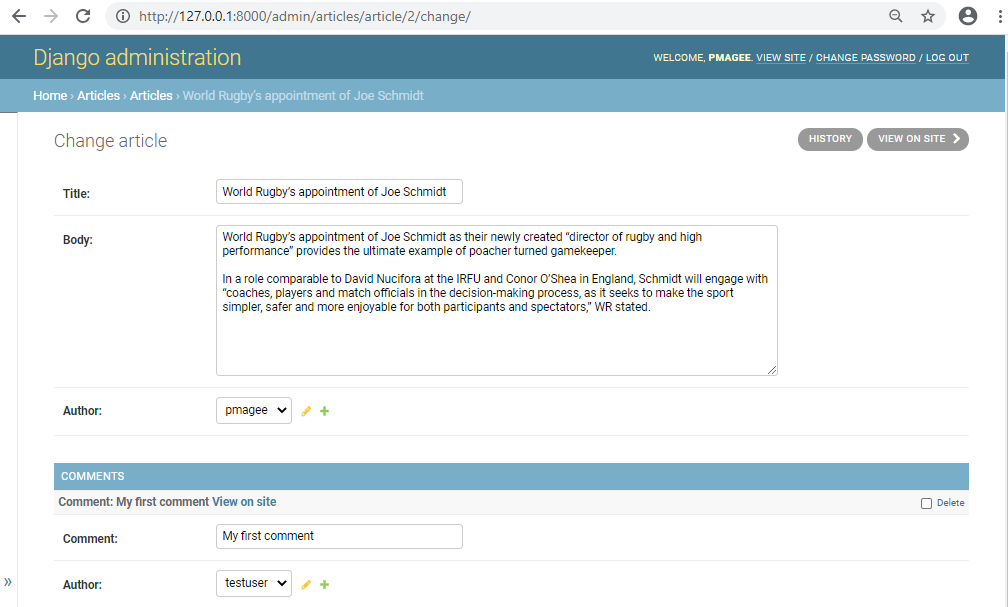
all model fields appear on one line while in a StackedInline each field has its own line.

We will implement both so you can decide which one you prefer.

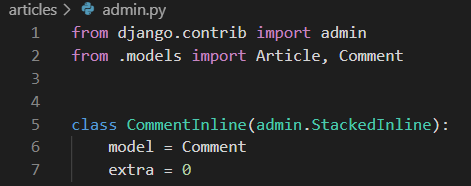
Update articles/admin.py as with the code shown below:



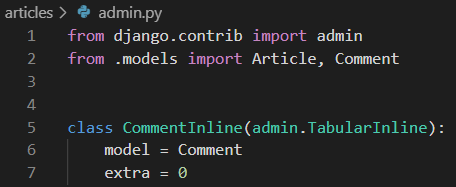
Now go back to the main admin page at http://127.0.0.1:8000/admin/ and click on “Articles.” Select the article which you just added a comment for which was “World’s Rugby Appointment” in the screen shown here.



We can see and modify all our related articles and comments in one place. Note that by default, the Django admin will display 3 empty rows here. You can change the default number that appear with the extra field. Add the line of code shown highlighted below:

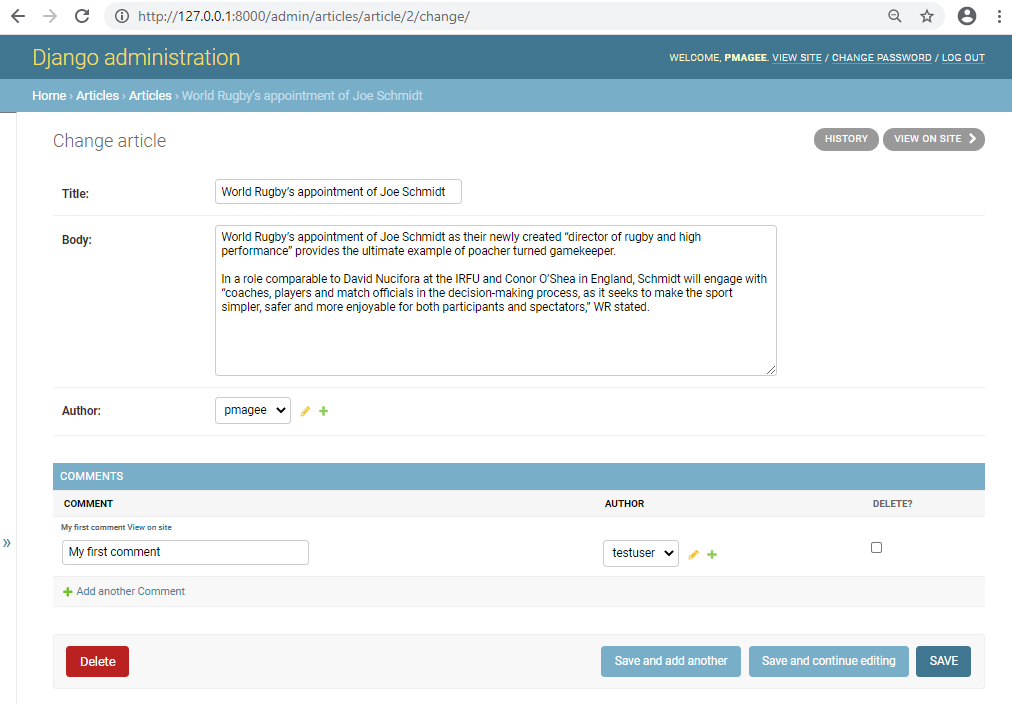


To switch to TabularInline change the CommentInline from admin.StackedInline to admin.TabularInline as shown in line 5 below:



Refresh the admin page and you’ll see the new change: all fields for each model are displayed on the same line.

**TabularInline**



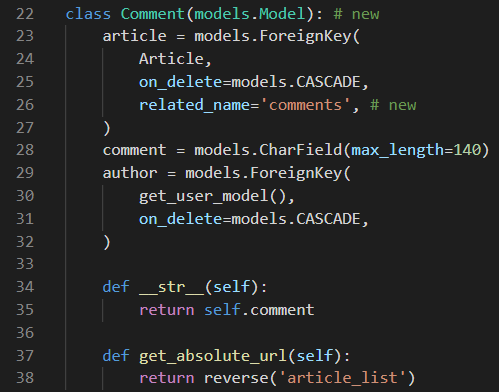
**Template**

Since Comment is part of our existing articles app we only need to update the existing templates for article\_list.html and article\_detail.html to display our new content. We don’t have to create new templates and mess around with URLs and views.

What we want to do is display all comments related to a specific article. This is called a “query” as we are asking the database for a specific bit of information. In our case, working with a foreign key, we want to follow a relationship backward: for each Article look up related Comment models.

In order to do this we will add a related\_name attribute to our model which lets us explicitly set the name of this reverse relationship.

To start, add a related\_name attribute to our Comment model. A good default is to name it the plural of the model holding the ForeignKey.



In our article\_list.html file we can add our comments to the card-footer. Note that I’ve moved the edit and delete links up into card-body. To access each comment we are calling article.comments.all which means first look at the article model, then comments which is the related name of the entire Comment model, and select all included. It can take a little while to become accustomed to this syntax for referencing foreign key data in a template!

{% extends 'base.html' %}

{% block title %}Articles{% endblock title %}

{% block content %}

{% for article in object\_list %}

<div class="card">

<div class="card-header">

<span class="font-weight-bold">{{ article.title }}</span> &middot;

<span class="text-muted">by {{ article.author }} | {{ article.date }}</span>

</div>

<div class="card-body">

<!-- Changes start here! -->

<p>{{ article.body }}</p>

<a href="{% url 'article\_edit' article.pk %}">Edit</a> |

<a href="{% url 'article\_delete' article.pk %}">Delete</a>

</div>

<div class="card-footer">

{% for comment in article.comments.all %}

<p>

<span class="font-weight-bold">{{ comment.author }} &middot;</span>

{{ comment }}

</p>

{% endfor %}

</div>

<!-- Changes end here! -->

</div>

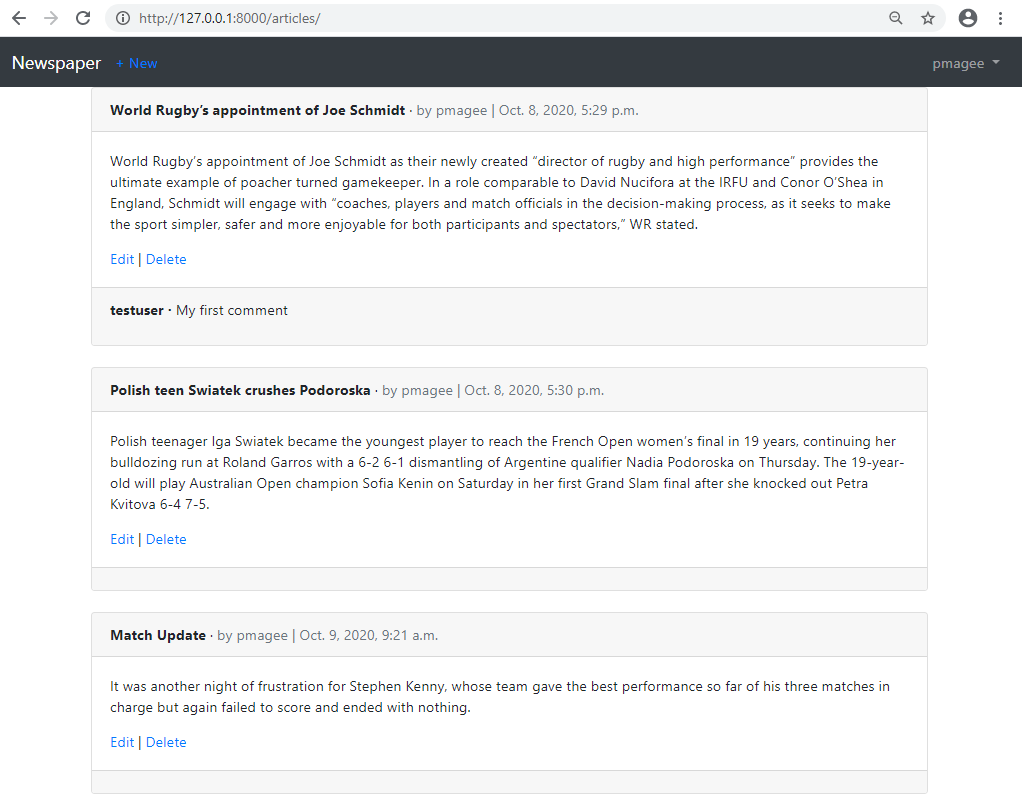
<br />

{% endfor %}

{% endblock content %}

If you refresh the articles page at http://127.0.0.1:8000/articles/ we can see our new

comment displayed on the page.



Run the following git commands to update the local and remote repositories:

(env) djangoprojects\lab5>git add -A

(env) djangoprojects\lab5>git commit -m “lab 5 part 5 commit”

(env) djangoprojects\lab5>git push -u origin master