Authorization, Pagination, and queryset in Django

Chapter 9

1

Authorization

- The previous chapter discussed on authentication. Authentication is the process of registering and logging-in users.
- Authorization restricts access. We might want to limit access to various pages only to logged-in users.

2

LoginRequiredMixin

- Restricting view access is just a matter of adding LoginRequiredMixin at the beginning of the views.
- For example, to limit that only logged-in users can add new post, we import LoginRequiredMixin in the views.py and add it in front of CreateView.

```
# views.py
from django.contrib.auth.mixins import LoginRequiredMixin
...
class ArticleCreateView(LoginRequiredMixin, CreateView):
...
```

- 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.

3

login_url

- A logged-out user, on the attempt to access a URL that is mapped to a view with LoginRequiredMixin, will be automatically redirected to the default location for the login page which is at /accounts/login.
- In case your login page is not at the default location, we can use "login_url" to indicate our login page.

```
# views.py
...
class ArticleCreateView(LoginRequiredMixin, CreateView):
        model = models.Article
        template_name = 'article_new.html'
        fields = ['title', 'body',]
        login_url = 'login' #
```

4

Pagination

- For example, on listing the posts, we want to add pagination so that we only list 2 posts on each page. This can be done with setting "paginate_by" attribute in the view.
- This limits the number of objects per page and adds a paginator and page_obj to the context.

```
class ArticleListView(LoginRequiredMixin, ListView):
    model = models.Article
    template_name = 'article_list.html'
    paginate_by = 2
    login_url = 'login'
```

5

paginator and page_obj

Having set "paginate_by" attribute in the view, we can then make use of the paginator and page_obj in our template files, such as:

- page_obj.has_previous, page_obj.has_next: Boolean
- page_obj.previous_page_number, page_obj.next_page_number : an integer
- page_obj.number: an integer, the current page numner
- page_obj.paginator.page_range
- page_obj.paginator.num_pages: an integer, the total number of pages

```
<div>
    {% if page_obj.has_previous %}
        <a href="?page=1">&laquo; first</a>
        <a href="?page={{ page_obj.previous_page_number }}">previous</a>
    {% endif %}

    {% for page in page_obj.paginator.page_range %}
        {% if page == page_obj.number %}
            {{page}}
        {% else %}
            <a href="?page={{page}}">{{page}}</a>
        {% endif %}
    {% endfor %}

    {% if page_obj.has_next %}
        <a href="?page={{ page_obj.next_page_number }}">next</a>
        <a href="?page={{ page_obj.paginator.num_pages }}">last &raquo;</a>
    {% endif %}
</div>
```

6

Filtering with queryset

- Instead of listing all the posts, we can use queryset to filter to only display posts with a published flag set to True.

```
class PostDetailView(DetailView):
    model = Post
    queryset = Post.objects.filter(published=True)
```

- As a matter of fact, in our views, specifying model = Post is really just shorthand for saying queryset = Post.objects.all()

7

Filtering by over-riding get_queryset

- Alternatively, we can go one step further and override the get_queryset method and use different querysets based on the properties of the request:

```
class PostDetailView(DetailView):
    model = Post
    def get_queryset(self):
        if self.request.GET.get("show_drafts"):
            return Post.objects.all()
        else: return Post.objects.filter(published=True)
```

8