Forms

Chapter 6

Objectives

- In this chapter we'll continue working on our blog application by adding forms so a user can create, edit, or delete any of their blog entries.

Forms

- Forms are very common and very complicated to implement correctly.
- Any time you are accepting user input there are
  - security concerns (XSS Attacks),
  - proper error handling is required, and
  - there are UI considerations around how to alert the user to problems with the form.
  - Not to mention the need for redirects on success.
- Fortunately, Django provides a rich set of tools to handle common use cases working with forms.

Adding a new post: View and URLConfs

- To start, update our base template to display a link to a page for entering new blog posts. It will take the form where post_new is the name for our URL.
- Let's add a new URLConf for post_new in the app-level urls.py file:

```
path('post/new/', views.BlogCreateView.as_view(), name='post_new'),
```

- Next, create our view by importing a new generic class called CreateView `from django.views.generic.edit import CreateView` and then subclass it to create a new view called BlogCreateView.

```
class BlogCreateView(CreateView):
    model = Post
    template_name = 'post_new.html'
    fields = '__all__'
```

Adding a new post: Template for creating the form

- The last step is to create our template, which we will call post_new.html.

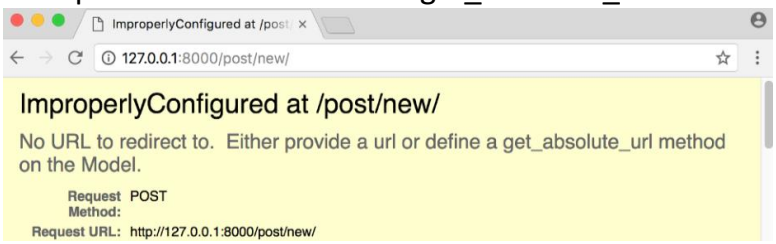
```
<!-- templates/post_new.html -->
{% extends 'base.html' %}
{% block content %}
    <h1>New post</h1>
    <form action="" method="post">{% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Save" />
        </form>
{% endblock %}
```

- Use HTML <form> tags with the POST method when sending data.
- For receiving data from a form, for example in a search box, use GET method.
- {% csrf_token %} is provided by Django to protect our form from cross-site scripting attacks.
- {{ form.as_p }} renders our output within paragraph <p> tags.
- Finally specify an input type of submit and assign it the value "Save".

- Try to add a new post and save it. Oops! What happened?

"ImproperlyConfigured" Exception

- You got an "ImproperlyConfigured" Exception with the value "No URL to redirect to. Either provide a url or define a get_absolute_url method on the Model."



- It's complaining that we did not specify where to send the user after successfully submitting the form.
- Let's send a user to the detail page after success; that way they can see their completed post.

Fixing the "ImproperlyConfigured" Exception

- We can follow Django's suggestion and add a get_absolute_url to our model.
- get_absolute_url() method tells Django how to calculate the canonical URL for an object, which is "the official URL where this model is displayed".
- Avoid hard coding paths in your templates. The reason for this is that paths might change, and it will be a hassle to go through all your HTML and templates to find every single URL or path and update it manually. It makes your code much harder to maintain.

```
<!-- BAD template code. Avoid! -->
<a href="/university/{{ object.id }}/">{{ object.full_name }}</a>

<!-- Correct -->
<a href="{{ object.get_absolute_url }}/">{{ object.full_name }}</a
```


Fixing the "ImproperlyConfigured" Exception

- In short, you should add a get_absolute_url() and __str__() method to each model you write.
- Open the models.py file. Add an import on the second line for reverse `from django.urls import reverse` and a new get_absolute_url method.

```
def get_absolute_url(self):
    return reverse('post_detail', args=[str(self.id)])
```

- Reverse is a utility function to reference an object by its URL template name, in this case "post_detail".

Fixing the ImproperlyConfigured" Exception (cont'd)

- Recall our URL pattern for "post_detail":

```
path('post/<int:pk>/', views.BlogDetailView.as_view(), name='post_detail'),
```

- That means in order for this route to work, we must pass in an argument with the primary key of the object.
- Django docs recommend using self.id with get_absolute_url.
- So we're telling Django that the ultimate location of a Post entry is its post_detail view which is post/<int:pk>/ so the route for the first entry we've made will be at posts/1.
- Try to create a new blog post again. Upon success, you are now redirected to the detailed view page where the post appears

Updating a post

- Let's use a built-in Django class-based generic view, UpdateView, to create the necessary template, url, and view for creating an update form so users can edit blog posts.
- To start, let's add a new link to post_detail.html so that the option to edit a blog post appears on an individual blog page.

```
<a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a>
```

- Next, we have to work on the view, url, and template. You should be familiar with this pattern now.

Updating a post: View and URLConfs

- **Edit** the application-level URLConfs to add a new URLConf for post_edit.

  path('post/<int:pk>/edit/', views.BlogUpdateView.as_view(), name='post_edit'),

- Create our view by importing a new generic class called UpdateView and then subclass it to create a new view called BlogUpdateView.

```
from django.views.generic import UpdateView
from .models import Post
class BlogUpdateView (UpdateView):
    model = Post
    template_name = 'post_edit.html'
    fields = ['title', 'body']
```

- Note that we are explicitly listing the fields ['title', 'body'] rather than using '__all__' because we assume that the author of the post is not changing.

Updating a post: Template to create the form

- **Create** the template file.

```
<!-- templates/post_edit.html -->
{% extends 'base.html' %}
{% block content %}
<h1>Edit post</h1>
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
<input type="submit" value="Update" />
</form>
{% endblock content %}
```

- When you try to edit a post on the browser, note that the form is pre-filled with the existing database data for the post.
- Make a change and after clicking the "Update" button, you are redirected to the detail view of the post where you can see the change.
- This is because of our get_absolute_url setting.

Deleting a post

- The process for creating a form to delete blog posts is very similar to that for updating a post. We'll use yet another generic class-based view, DeleteView, to help and need to create a view, url, and template for the functionality.

Deleting a post: View and URLConfs

- **Edit** the application-level URLConfs to add a new URLConf for post_delete.
  path('post/<int:pk>/delete/', views.BlogDeleteView.as_view(), name='post_delete'),
- Create our view by importing a new generic class called DeleteView and then subclass it to create a new view called BlogDeleteView.
- Note the use of reverse_lazy for success_url.
- In add new post and update post, on success, you are redirected to the detail view of the post where you can see the change. This is because of our get_absolute_url setting in the model.

```
from django.views.generic import DeleteView
from django.urls import reverse_lazy
from .models import Post
class BlogDeleteView (DeleteView):
    model = Post
    template_name = 'post_delete.html'
    success_url = reverse_lazy('home')
```

- Here, reverse_lazy as opposed to just reverse (used get_absolute_url setting) so that it won't execute the URL redirect until our view has finished deleting the blog post. And we have indicated to go to our URL pattern for "home".

Deleting a post: Template to create the form

- **Create** the template file. Note we are using post.title here to display the title of our blog post and we give the value “Confirm” on the submit button

```
<!-- templates/post_delete.html -->
{% extends 'base.html' %}
{% block content %}
<h1>Delete post</h1>
<form action="" method="post">{% csrf_token %}
<p>Are you sure you want to delete "{{ post.title }}"?</p>
<input type="submit" value="Confirm" />
</form>
{% endblock content %}
```


Overview of generic views used

- Template View: to present some information in a html page.
- ListView: to present a list of objects in a html page.
- DetailView: to present detail of a single model instance.
- FormView: to present a form on the page and perform certain action when a valid form is submitted. eg: Having a contact us form and sending an email on form submission.
- CreateView: to present a form on the page and need to do a database insertion on submission of a valid form.
- UpdateView
- DeleteView

Summary: what you have learnt

- A generic class called CreateView, UpdateView, DeleteView
- Post method for form submission
- The use of {% csrf_token %} to protect the form from cross-site scripting attacks.
- Use the utility function “reverse” and “reverse_lazy” to reference an object by its URL template name