Forms and Static Files

Chapter 5

Objectives

- In this chapter we'll handle the CRUD (create, read, update, delete) operations of a model.
- Introduce class-based CreateView, UpdateView, DeleteView
- Introduce forms.py
- Introduce CSS for styling
- Learn how Django works with static files

Adding a new post: 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'),
```

- Our url will start with post/new/, the view is called BlogCreateView, and the url will be named post_new.
- Next, let's create a new view called BlogCreateView.

Adding a new post: View with class-based CreateView

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

```
from django.views.generic.edit import CreateView
class BlogCreateView(CreateView):
    model = Post
    template_name = 'post_new.html'
    fields = '__all__'
```

Here, we will display all the fields from the Post model

- Within BlogCreateView, we specify our database model Post, the name of our template post_new.html, and all fields with '__all__'.
- The use of the generic view CreateView automatically generates the Django form for us to add the records into the database model for Post, similar to executing the SQL statement 'INSERT INTO ...."

Adding a post: Template for creating the form

- The last step is to create our template, post_new.html.

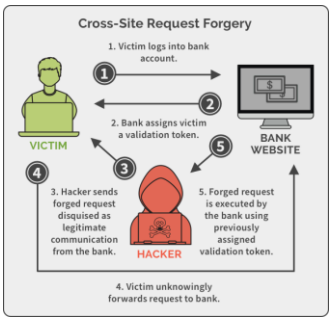
```
<!-- templates/post_new.html -->
{% extends 'base.html' %}
{% block content %}
    <h1>New post</h1>
    <form action="" method="post">{% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Save" />
        </form>
 {% endblock %}
```

- 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 request forgery.
- {{ form.as_p }} renders our output within paragraph <p> tags.
- Finally specify an input type of submit which is a button and assign "Save" as the button caption.

Cross Site Request Forgery protection

- CSRF is a type of Cross Site Scripting attack.
- It occurs when a malicious website contains a link, a form button or some JavaScript that is intended to perform some action on your website, using the credentials of a logged-in user who visits the malicious site in their browser.
- **Django** protects against **CSRF** attacks by generating a **CSRF** token in the server, send it to the client side via a hidden field, and requesting the client to send the token back in the request header. The server will then verify if the token from client is the same as the one generated previously; if not it will not authorise the request.
- A **CSRF token** is a unique, secret, unpredictable value that is **generated** by the server-side application and transmitted to the client in such a way that it is included in a subsequent HTTP request made by the client.

CSRF attack scenario



- In the attacker's ideal situation, the target victim may perform monetary transactions online with a bank whose website is vulnerable to CSRF.
- The attacker, in targeting all customers of this particular bank, has set up a malicious website that, when navigated to by the still-logged-in victim, will transfer a sum of money to the attacker's account.
- The malicious link may take the form of an image, banner ad, or even a site that replicates the bank's own website.
- An attacker will typically embed HTML or JavaScript code into the link which will request a specific task – in this scenario, the task involves the transfer of funds to the attacker's account.
- With the request originating from the browser of an authenticated user, the online bank processes the task as requested.
- A 12-minute demo using Flask: A good example demo using Flask -- https://youtu.be/TNM0X7Hmv0E

"ImproperlyConfigured" Exception

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

get_absolute_url() method of a model

- Following Django's suggestion, add a get_absolute_url to our model to send a user to the detail page after successful insertion of a record.
- Open the models.py file. Add an import on the second line for reverse

```
from django.urls import reverse
```

- Then, add a new get_absolute_url method. Remember to pay attention to proper indentation.

```
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", and we need to pass the primary key (self.id) in order to load the detail of the chosen post.

URL pattern for post_detail with a primary key

- Our URL pattern for "post_detail" with a primary key to locate a specific record:

```
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. So the route for the first entry will be at post/1.
- Let's write BlogDetailView and the corresponding template to display the details of a specific record.

BlogDetailView and the template file

```
class BlogDetailView (DetailView):
    model = Post
    template_name = 'post_detail.html'
```

```
<!-- templates/post_detail.html -->
{% extends 'base.html' %}
{% block content %}
    <h2>{{ post.title }}</h2>
    <p>{{ post.body }}</p >
    <a href="{% url 'post_edit' post.pk  %}">+ Edit Blog Post</a>
    <p><a href="{% url 'post_delete' post.pk %}">+ Delete Blog Post</a></p>
{% endblock content %}
```

- Try to create a new blog post again. Upon success, you are redirected to the detailed view page where the post appears.

Django's DetailView

- Django's class-based DetailView refers to a view (logic) to display one instance of a table in the database.
- By default, it expects the pk as argument for the generic view.
- We can override get_object() so that it gets the desired object from the database, e.g.

```
class TicketDetail(DetailView):
    model = Ticket

    def get_object(self, queryset=None):
        return Ticket.objects.get(uuid=self.kwargs.get("uuid"))
```

Returning extra data with a generic view

- get_context_data()

```
class AppDetailsView(generic.DetailView):
  model = Application
  def get_context_data(self, **kwargs):
      context = super().get_context_data(**kwargs)
      context['category'] = "MISC"
      return context
```

```
<h1>{{ object.title }}</h1>
<p>{{ object.description }}</p>
<p>{{ category }}</p>
```


Editing/Updating a post

- Let's use a built-in Django class-based generic view, UpdateView, similar to executing the SQL statement "UPDATE Table SET …" 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.

- Note tha `<a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a>` which post to display the detail.
- Next, we have to work on the view, url, and template. You should be familiar with this pattern now.
- **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'),

Editing a post: View with class-based UpdateView

- Create our view by importing a new generic class called UpdateView and then subclass it to create a new view called BlogUpdateView.
- The use of the generic view UpdateView automatically generates the Django form to edit the data of a particular record from the database
model for Post.

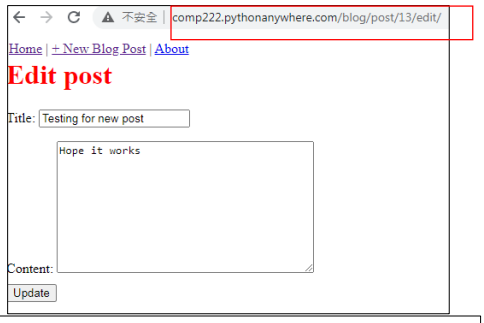
```
from django.views.generic import UpdateView
from .models import Post

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.

Editing a post: Template to create the form

- **Create** the template file post_edit.html as stated in BlogUpdateView.

```
<!-- 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 edit a post on the browser, the form is pre-filled with the existing database data for the post.
- Make a change and click 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 editing a post.
- Let's use a built-in Django class-based generic view, DeleteView, similar to executing the SQL statement "DELETE FROM Table…" for creating the form to delete a particular post.
- To start, let's add a new link to post_detail.html so that the option to delete a blog post appears on an individual blog page.

```
<p><a href="{% url 'post_delete' post.pk %}">+ Delete Blog Post</a>
```

- Note that we have to pass post.pk to the URL so that it knows which post to delete.
- Next, we have to work on the view, url, and template. You should be familiar with this pattern now.
- **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'),
```


Deleting a post: class-based DeleteView

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

```
from django.views.generic import DeleteView
from .models import Post
from django.urls import reverse_lazy

class BlogDeleteView (DeleteView):
    model = Post
    template_name = 'post_delete.html'
    success_url = reverse_lazy('home')
```

- Note the use of reverse_lazy for success_url.
- In adding new post and update post, on success, we are redirected to the details view of the post to see the change. This is because of our get_absolute_url setting in the model.
- After delete is performed, there is no details page of the deleted record to be shown anymore. Hence, we cannot take advantage of the get_absolute_url method that we have written in the Post model
- What we are doing here is to load the homepage after delete is done successfully.
- Here, reverse_lazy as opposed to just reverse (used in 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".

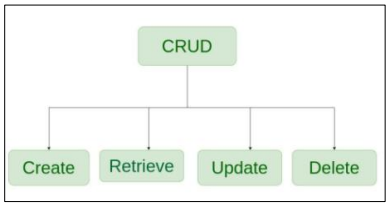
Deleting a post: Template to create the form

- **Create** the template file post_delete.html as stated in BlogDeleteView.
- 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 %}
```

Django CRUD (Create, Retrieve, Update, Delete) Class Based Views

- CreateView – create or add new entries in a table in the database.
- Retrieve Views – read, retrieve, search, or view existing entries as a list(ListView) or retrieve a particular entry in detail (DetailView)
- UpdateView – update or edit existing entries in a table in the database
- DeleteView – delete, deactivate, or remove existing entries in a table in the database
- TemplateView: to present some information in a html page.



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 actually shorthand for saying queryset = Post.objects.all()

OR

```
class PostDetailView(DetailView):
    …
    def get_queryset(self):
        return models.Post.objects.filter(published=True)
```

filter() method returns a QuerySet, which is like a list.

21

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)
```

22

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.

form from generic class-based views

- On slide 4, in our class-based BlogCreateView, we have used the generic CreateView to present a form to do a database insertion on submission of a valid form.
- Similarly, on slide 13, in our class-based BlogUpdateView, we used the generic UpdateView to present a form to update the data in our database.
- As a result, in our template files, we can use {{ form.as_p}} :

```
<form action="" method="post">{% csrf_token %}
  {{ form.as_p}}
<input type="submit" value="Save" />
</form>
```

Introducing forms.py

- Django comes with a form library, called django.forms, that handles HTML form display with validation.
- The primary way to use the forms framework is to define a Form class for each HTML <form> you're dealing with.
- By convention, keep this Form class in a file called forms.py, in the **<u>same directory as your models.py.</u>**

forms.py - forms.Form

- On slide 4, in our class-based BlogCreateView, we have used the generic CreateView to present a form to do a database insertion on submission of a valid form.
- We can build your form that is not hooked to a database with forms.Form
- The ContactForm shown below is an example.

forms.py

ContactForm clean_message

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False, label='Your e-mail address')
    message = forms.CharField(widget=forms.Textarea)
```

Will render <textarea> as html

Tying form objects into views

```
# views.py

from django.shortcuts import render
from mysite.forms import ContactForm
from django.http import HttpResponseRedirect
from django.core.mail import send_mail

# ...

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render(request, 'contact_form.html', {'form': form})
```

Form is valid if all the required fields contain data.

We will use POST method to submit data from the client.

A cleaned_data is a dictionary of the submitted data, "cleaned up" by converting values to the appropriate Python types.

This is an empty form. You can later try the following to note the difference:
form = ContactForm(**initial={'subject': 'I like your site!'}**)

What is the result of render()?

render(*request*, *template_name*, *context=None*)

- Combines a given template with a given context dictionary and returns an **HttpResponse** object with that rendered text.
- ***request*** and ***template_name*** are required arguments.
- Optional argument:
  - **context:** A dictionary of values to add to the template context. By default, this is an empty dictionary.

Create our contact form

- Create the template, contact_form.html
- And finally, change our urls.py to display our contact form at /contact/.

contact_form.html

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>
    {% if form.errors %}
        <p style="color: red;">
        Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}
    <form action="" method="post">
        <table>
            {{ form.as_table }}
        </table>
        {% csrf_token %}
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

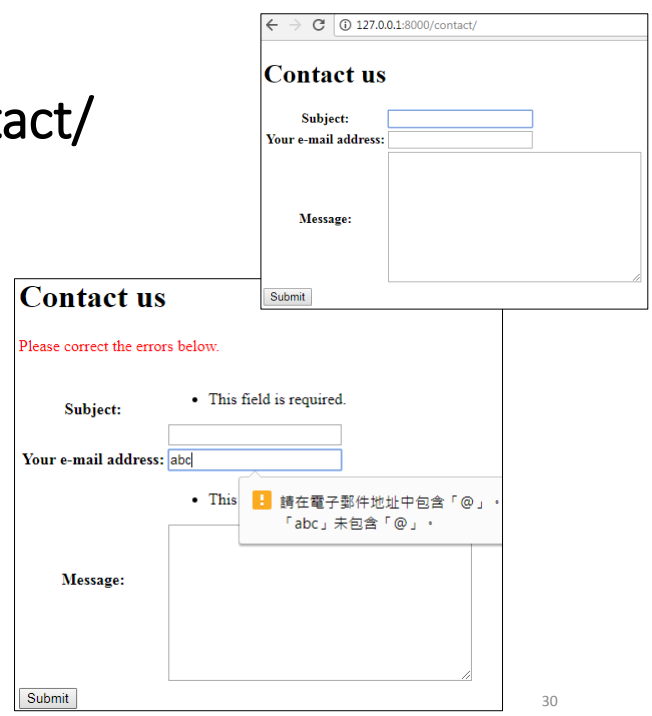
Other options are as_ul(), as_p()

29

Result of running /contact/

Load the form,

- submit it with none of the fields filled out,
- submit it with an invalid e-mail address, then finally
- submit it with valid data.
  - Of course, unless you have configured a mail-server, you will get a ConnectionRefusedError when send_mail() is called.



30

Custom validation

- Django's form system automatically looks for any method whose name **starts with clean_ and ends with the name of a field**.
- If any such method exists, it's called during validation.
- Specifically, the clean_message() method will be called after the default validation logic for a given field (in this case, the validation logic for a required CharField).

forms.py

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)

    def clean_message(self):
        message = self.cleaned_data['message']
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

If we forget the return statement, then None will be returned, and the original value will be lost.

We don't have to worry about checking that the value exists and is non-empty; that's done by the default validator

Django Forms vs Plain HTML Forms

Advantages of Django forms:

- Django's form class provides validation on many forms. For example, if you create a form and there are many blank fields (by default all fields are required meaning a user must enter in a value), the form will be submitted; the user will be told to fill in the blank fields before the form is submitted.
- Django's form class also provides validation with many different types of data entry. For example, it has an EmailField that validates email address. So, if a user types in, 'Peter' as his email, Django's form states that this is not a valid email address.
- Django forms gives ease of use when taking data from a form and inserting the data into a database.

Django Forms vs Plain HTML Forms (cont'd)

Advantages of Plain HTML Forms

- Using plain HTML forms, you are not restricted to Django's form class rules, so you have more flexibility to make the form however you want.

Disadvantages of Plain HTML Forms

- So the disadvantages of plain HTML forms is that you have to write all of the validation tools yourself. This includes email validation, integer validation (if working with numbers), etc.

Static files

- CSS is referred to as a static file because, unlike our dynamic database content, it doesn't change.
- First create a project-level folder called static.
- Just as we did with our templates folder, we need to update settings.py to tell Django where to look for these static files.
- We can update settings.py with a one-line change for STATICFILES_DIRS. Add it at the bottom of the file below the entry for STATIC_URL.

```
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

STATIC_URL vs STATICFILES_DIRS

- With STATICFILES_DIRS, we tell Django where to look for these static files.
- Why do we still need STATIC_URL? This is for directly accessing the static files via the URL.
- For example, if you have a static file stored at home/blogproject/static/img/1.png, the following URL access will cause an error:

  http://username.pythonanywhere.com/home/blogproject/static/img/1.png
- With STATIC_URL = '/static/', the correct way should be: http://username.pythonanywhere.com/static/img/1.png
- In other words, http://username.pythonanywhere.com/static/ is mapped to the location as specified in STATICFILES_DIRS.

Static files (cont'd)

- Now create a css folder within static and add a new base.css file in it.
- Let's use this css file to change the title to be red.

```
/* static/css/base.css */

header h1 a {

  color: red;

}
```

- We need to add the static files to our templates by adding {% load static %} to the top of base.html.
- Since the other templates inherit from base.html, we only have to add this once.
- Include a new line at the bottom of the <head></head> code that explicitly references our new base.css file.

Static files (cont'd)

```
<!-- templates/base.html -->
{% load static %}
<html>
  <head>
    <title>Django blog</title>
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
  </head>
  ...
```

- {% static %} template filter will add the URL specified in STATIC_URL in front, hence, it will become /static/css/base.css
- Start up the server again and look at our updated homepage.

Summary: what we have learnt

- Generic classes: CreateView, UpdateView, DeleteView -- in our class-based views, the use of these generic views automatically generate the Django forms for us to create, edit and delete the records from the database model respectively.
- 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.
- In creating new post and update post, on success, we use get_absolute_url setting in the model to use the utility function "reverse" to redirect to the detail view of the post where we can see the change .
- In deleting a post, reverse_lazy instead of reverse is used so that it won't execute the URL redirect until our view has finished deleting the blog post.