30 December 2019

Django Shortcuts and Decorators

1. **Render** : Combines a given template with a given context dictionary and returns an HttpResponse object with that rendered text

render(request, template_name, context=None, content_type=None, status=None, using =None)

request

The request object used to generate this response.

template_name

The full name of a template to use or sequence of template names. If a sequence is given, the first template that exists will be used.

context

A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the view will call it just before rendering the template.

status

The status code for the response. Defaults to **200**.

2. **get_object_or_404()**

get_object_or_404(*class*, **args*, ***kwargs)*

Calls **get()** on a given model manager, but it raises **Http404** instead of the model's **DoesNotExist** exception

Django Built-In Views

1. The 404 Page not found View

When you raise *Http404* from within a view, Django loads a special view devoted to handling 404 errors. By default, it's the view *django.views.defaults.page_not_found()*, which either produces a "Not Found" message or loads and renders the template 404.html if you created it in your root template directory.

The default 404 view will pass two variables to the template: request_path, which is the URL that resulted in the error, and exception, which is a useful representation of the exception that triggered the view (e.g. containing any message passed to a specific Http404 instance).

Request and Response Objects

Django uses request and responses to pass information through the system. When a page is requested, Django creates a HttpRequest that contains data about the request for example who made the. request, any csrf tokens etc.

Django will then match the HttpRequest with the urlpatterns and then load the first matching pattern's view. The request is passed as the first argument to the function. The view is then executed and the view should return a HttpResponse object.

HttpRequest Object

Has a tonne of attributes, however the main ones are

1. Path : A string representing the full path to the requested page, not including the scheme or domain.

2. Method : A string representing the HTTP method used in the request. This is guaranteed to be uppercase. For example:

```
if request.method == 'GET':
    do_something()
elif request.method == 'POST':
    do_something_else()
```

Getting data passed :

All non-form data can be accessed from the request.body

All form data can be from the request.POST in the form of dictionary.

However files cannot be accessed via the above two methods.

All file uploads are done via the request.files. It is a dictionary containing all the uploaded files

FILES will only contain data if the request method was POST and the <form> that posted to the request had **enctype="multipart/form-data"**. Otherwise, FILES will be a blank dictionary-like object.

Attributes set by the middleware

HttpRequest.**user**

From the *AuthenticationMiddleware*: An instance of *AUTH_USER_MODEL* representing the currently logged-in user. If the user isn't currently logged in, user will be set to an instance of *AnonymousUser*. You can tell them apart with *is_authenticated*, like so:

QueryDict

When the get and post request return some information, the returned data is in the form of a querydict which is very similar to python dictionary.

HttpResponse Object

In contrast to *HttpRequest* objects, which are created automatically by Django, *HttpResponse* objects are your responsibility. Each view you write is responsible for instantiating, populating, and returning an *HttpResponse*.

The *HttpResponse* class lives in the *django.http* module.

Sometimes instead of returning a template we would want to return a file. This is how we tell Django that the response object is a MS Excel file :

```
>>> response = HttpResponse(my_data, content_type='application/vnd.ms-excel') >>>
response['Content-Disposition'] = 'attachment; filename="foo.xls"'
```

It is not possible always to have a basic HttpResponse. So Django subclasses HttpResponse for our usage :

1. HttpResponseRedirect : The first argument of this constructor is the path to redirect to. It can be a full path or even a relative path
2. HttpResponsePermanentRedirect : Similar to the above one except returns a status code of 302 instead if 301

FileResponse Objects

Subclass of StreamingHttpResponse optimised for binary files.

class FileResponse(*open_file,as_attachment=False,filename=”,**kwargs*)
FileResponse is a subclass of *StreamingHttpResponse* optimised for binary files. It uses wsgi.file_wrapper if provided by the wsgi server, otherwise it streams the file out in small chunks.

If as_attachment=True, the Content-Disposition header is set to attachment, which asks the browser to offer the file to the user as a download. Otherwise, a Content-Disposition header with a value of inline (the browser default) will be set only if a filename is available.

If open_file doesn’t have a name or if the name of open_file isn’t appropriate, provide a custom file name using the filename parameter. Note that if you pass a

file-like object like io.BytesIO, it's your task to seek() it before passing it to FileResponse.

The Content-Length and Content-Type headers are automatically set when they can be guessed from contents of open_file.

FileResponse accepts any file-like object with binary content, for example a file open in binary mode like so:

```
>>> from django.http import FileResponse
>>> response = FileResponse(open('myfile.png', 'rb'))
```

TemplateResponse is responsible for delaying the rendering template until the entire view has been executed. Sometimes we may want this. We want to wait to see any middleware or some function changes the result before rendering a view.

```
>>> t = TemplateResponse(request, 'original.html', {})

>>> t.render()
```

File Handling

When a file is uploaded, the file is placed in the request.Files in the form of a dictionary. This also include ImageField which is a subclass of the FileField.

form = UploadFileForm(request.POST, request.FILES)

We have to pass the request.Files to the form constructor.

If we are saving a file on a model, then modelForm makes it easier for us.

However multiple file upload handling is a bit different.

The *django.core.files* module and its submodules contain built-in classes for basic file handling in Django.

Managing Files

By default, Django stores files locally, using the *MEDIA_ROOT* and *MEDIA_URL* settings.

```
class Car(models.Model):
        name = models.CharField(max_length=255)
        price = models.DecimalField(max_digits=5, decimal_places=2)
        photo = models.ImageField(upload_to='cars')
```

```
>>> car = Car.objects.get(name="57 Chevy")
```

```
>>> car.photo
<ImageFieldFile: cars/chevy.jpg>
```

Some methods are :

1. Name : cars/chevy.jpg
2. Path. :'/media/cars/chevy.jpg'
3. Url : gives us the url of how to access the file.

Django deals with all file types using the in-built django.core.files.Files

Class Based Views

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case.

Because Django's URL resolver expects to send the request and associated arguments to a callable function, not a class, class-based views have an *as_view()* class method which returns a function that can be called when a request arrives for a URL matching the associated pattern.

When as_view() is called an instant of that class is created and then its attributes are initialised using setup(). After that dispatch method is called which determines which method to call (get or post) based on the request header.

path('about/', MyView.as_view()),

Sometimes we may want to have some class attributes which are accessible in all the methods

```
from django.http import HttpResponse from django.views import View

class GreetingView(View):

        greeting = "Good Day"

        def get(self, request):
                return HttpResponse(self.greeting)
```

This can be easily overwritten in the subclass by just having a greeting in the class attribute of the subclass.

It is possible to apply decorators on classes we can do it in the urls.py file.

```
from django.contrib.auth.decorators import login_required, permission_required

from django.views.generic import TemplateView

from .views import VoteView

urlpatterns = [
        path('about/', login_required(TemplateView.as_view(template_name="secret.html"))),
        path('vote/', permission_required('polls.can_vote')(VoteView.as_view())),

]
```

Decorating the class can be done by applying the decorator on the dispatch() method

```
class ProtectedView(TemplateView):

template_name = 'secret.html'

        @method_decorator(login_required)

        def dispatch(self, *args, **kwargs):

                return super().dispatch(*args, **kwargs)
```

OR

```
@method_decorator(login_required, name='dispatch')

Class ProtectedView(TemplateView)
```

Sometimes we may want to display the model instances that we have in the form of a list. Then we can use the list view.

Writing model = “model name” is same as writing

```
queryset = modelName.objects.all()
```

Even these can be added as class attribute

```
context_object_name = 'book_list'
queryset = Book.objects.filter(publisher__name='ACME Publishing')

context_object_name = 'book_list'

template_name = 'books/acme_list.html'
```

```
# urls.py

from django.urls import path
from books.views import PublisherBookList

urlpatterns = [
        path('books/<publisher>/', PublisherBookList.as_view()),

]
```

```
# views.py

from django.shortcuts import get_object_or_404 from django.views.generic import
ListView
from books.models import Book, Publisher

class PublisherBookList(ListView):

        template_name = 'books/books_by_publisher.html'

        def get_queryset(self):
                self.publisher = get_object_or_404(Publisher, name=self.kwargs['publisher'])
                return Book.objects.filter(publisher=self.publisher)
```

To add a new context

```
def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context

        context = super().get_context_data(**kwargs)
        # Add in a QuerySet of all the books

        context['book_list'] = Book.objects.all()
        return context
```

API Reference for CBVs

Base Views

1. View :

   1.1.Attributes : http_method_names : Allowed Http Method names

2. TemplateView : Renders a given template, with the context containing parameters captured in the URL.

   2.1 : Attributes : template_name : name of html file to be rendered

   extra_content : To pass some extra information

3. RedirectView : Redirects to a given URL.

   3.1. Attributes : url : The url to redirect to as a string

   pattern_name : The name of the URL pattern to redirect to. Reversing will be done using the same args and kwargs as are passed in for this view.

   permanent : Whether the redirect should be permanent.

Generic Display Views

1. DetailView : Can only be used when he have pass some sort of information via the URL in the form of kwargs.
2. ListView : Used to display list of objects.

Generic Editing Views

1. FormView : A view that displays a form. On error, redisplays the form with validation errors; on success, redirects to a new URL.
2. CreateView : Used to create an object and if required redisplaying the form with validation error
3. UpdateView : A view that displays a form. On error, redisplays the form with validation errors; on success, redirects to a new URL.

4. DeleteView : A view that displays a confirmation page and deletes an existing object. The given object will only be deleted if the request method is POST. If this view is fetched via GET, it will display a confirmation page that should contain a form that POSTs to the same URL.

Generic Date Views

Used to display data based on the date-field

Class Based Views Mixins

1. ContentMixin : Add extra content to the context dictionary
2. TemplateResponseMixin : Allows us to create a TemplateResponse (Used to render templates)
3. SingleObjectMixin : Provides a mechanism for looking up an object associated with the current HTTP request.
4. SingleObjectTemplateResponseMixin : A mixin class that performs template-based response rendering for views that operate upon a single object instance.
5. MultipleObjectMixin : Used to display a list of objects [Similarly we have MultipleObjectTemplateResponseMixin]
6. FormMixin : Create and display forms
7. ModelFormMixin : Works on normal model form rather normal forms

reverse()

Allows us to get the url to be fetched by using the namespace.

reverse(*viewname*, *urlconf=None*, *args=None*, *kwargs=None*, *current_app=None*)

```
from django.urls import reverse

def myview(request):
        return HttpResponseRedirect(reverse('arch-summary', args=[1945]))
```