**Deep-dive Django views**

[[Marshal Great](https://great-devxy.medium.com/?source=post_page-----4aab0e8d289c--------------------------------)](https://great-devxy.medium.com/?source=post_page-----4aab0e8d289c--------------------------------)

[Marshal Great](https://great-devxy.medium.com/?source=post_page-----4aab0e8d289c--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F8007ad70d030&operation=register&redirect=https%3A%2F%2Fgreat-devxy.medium.com%2Fdeep-dive-django-views-4aab0e8d289c&user=Marshal+Great&userId=8007ad70d030&source=post_page-8007ad70d030----4aab0e8d289c---------------------post_header-----------)

7 min read

·

Jul 16, 2023

8



Django views are in charge of providing users with data for viewing and consumption. In Django’s MVT framework, as we’ve discussed in our last [*article* *here*](https://medium.com/@great-devxy/what-is-django-mvt-architecture-what-differs-mvt-from-mvc-architecture-f80cec80b05), views are a critical element. Views are essentially Python functions that take an HTTP request as an argument and output a suitable response. In Django’s views, the logic underlying the entire request-and-response cycle of web applications is stored.

The view function is a Python function that accepts a web request and returns a web response, according to the Django documentation. The solution might consist of HTML code for a web page, a 404 error, a redirect, an image, an XML document, or something different. The user interface then shows the webpage’s content when the website has been rendered. It is expressed using HTML, CSS, and Javascript. The view functions are included in the views.py file, which may be located in the Django application’s folder.

**Types Of Django Views**

In order to learn Django, you have to understand the concept that a view is just a function. It’s essential to comprehend how they work and how your application fits inside them. We go more into the realm of Django views in this part. We have two types of Django views, namely:

1. Function-Based View (FBV):
2. Class-Based View (CBV):

**Function-Based View (FBV):**

Function-based views are simple and easy to utilize, particularly for small views or rapid prototypes. They may, however, become more difficult to maintain as the complexity of your views increases. In such circumstances, class-based views, which allow more flexibility and reusability, may be considered.In current projects, function-based views are still fully supported and actively utilized.

from django.http import HttpResponse  
  
 def hello\_world(request):  
 return HttpResponse("Hello, World!")

from django.shortcuts import render  
  
def product\_detail(request, product\_id):  
 product = Product.objects.get(id=product\_id)  
 context = {'product': product}  
 return render(request, 'product\_detail.html', context)

**Advantages of Function-Based Views (FBV)**

1. Django offers a number of quick cuts to make creating FBVs simpler. Above is an example of a little more complicated view that returns a HttpResponse with a template using the render shortcut function.
2. Forms, sessions, generic display, edit views, and any other Django functionality may all be handled via FBVs.

However, you may want to consider class-based views for better structure and reuse of code. As your project expands, they may make your code more modular and manageable. But before going on to class-based views, having a solid knowledge of FBVs is a wonderful starting point.

2. **Class-Based View (CBV):**

Django’s Class-Based Views (CBVs) provide an alternate method for constructing views that uses Python objects rather than functions. They were created specifically to make it simple to reuse code and arrange it in a more logical, modular fashion.

Because CBVs utilize Python’s inheritance paradigm to function, you can simply expand and replace them. As a result, you may create several generic and reusable views and cut down on the amount of code required to implement common patterns. Django has a variety of built-in CBVs, including:

1. **View:** The basic class for all other views is this one. It shows a callable object that receives a web request and sends back a web response. This answer might consist of a document’s HTML content, a redirect, a 404 error, an XML file, an image, or anything else. This basic view’s helpful methods, such as as\_view(), dispatch(), http\_method\_not\_allowed(), and others, may be inherited by subclassing it.

from django.http import HttpResponse  
from django.views import View  
  
class MyView(View):  
 def get(self, request):  
 return HttpResponse('Hello, World!')

2. **TemplateView:**To display a given template and provide a response, it uses a straightforward class-based view. It’s often used for static pages with little logic, including about and contact pages.

from django.views.generic import TemplateView  
  
class HomePageView(TemplateView):  
 template\_name = 'home.html'

3. **ListView:**ListView offers a simple method for displaying model data as a list. It is a general, high-level view for listing several things.

from django.views.generic.list import ListView  
from myapp.models import MyModel  
  
class MyModelListView(ListView):  
 model = MyModel

4. **DetailView:**When presenting a detailed view of a single model instance, the DetailView is utilized. A single item will be shown in this high-level, general view.

from django.views.generic.detail import DetailView  
from myapp.models import MyModel  
  
class MyModelDetailView(DetailView):  
 model = MyModel

5. **CreateView:**To generate an object, this view is utilized. It is a broad, high-level perspective for managing model items. It processes user input via a form and adjusts the model accordingly.

from django.views.generic.edit import CreateView  
from myapp.models import MyModel  
  
class MyModelCreateView(CreateView):  
 model = MyModel  
 fields = ['name', 'description']

6. **UpdateView:**An existing record in the database can be edited using this view. The existing object’s form is rendered, and the POST request to change it is handled.

from django.views.generic.edit import UpdateView  
from myapp.models import MyModel  
  
class MyModelUpdateView(UpdateView):  
 model = MyModel  
 fields = ['name', 'description']  
 template\_name\_suffix = '\_update\_form' # use a custom template name

7. **DeleteView:**To remove an existing record from the database, this view used. It handles the DELETE request to delete the object and renders a confirmation page.

from django.views.generic.edit import DeleteView  
from myapp.models import MyModel  
  
class MyModelDeleteView(DeleteView):  
 model = MyModel  
 success\_url = '/success/' # the URL to redirect to after successful deletion

8. **FormView:**Form management and submission are done using this view. It takes care of drawing the form for GET requests, re-rendering it with validation errors if the form is invalid, processing the data and re-directing the user for successful POST requests, and handling rendering the form for GET requests.

from django.views.generic.edit import FormView  
from django.http import HttpResponseRedirect  
from myapp.forms import MyForm  
  
class MyFormView(FormView):  
 form\_class = MyForm  
 template\_name = 'myform.html'  
 success\_url = '/success/'  
  
 def form\_valid(self, form):  
 # This method is called when valid form data has been POSTed.  
 # It should return an HttpResponse.  
 form.send\_email()  
 return super().form\_valid(form)

**Advantages of Class-Based View (CBV):**

1. Django’s class-based views can significantly speed up the process of developing reliable, scalable web applications.
2. They enable developers to benefit from modular, reusable components and spend less time writing repetitive code.
3. The extensive collection of generic views offered by Django addresses a wide range of typical use cases, from straightforward template rendering to sophisticated form handling.

**Best Practices for Django Views**

Even though Django views may seem easy, they are very flexible and can have a big impact on how scalable, reliable, and fast your web app is. Because of this, it is very important to follow some best practices when writing Django views. This article will go into detail about these best practices, explaining why they are important and how they can change the way you build Django applications.

1. **Keep Business Logic Out of Views:**The purpose of views in Django is to act as a bridge between the user and the system. They are in charge of getting HTTP requests, doing something with them, and sending HTTP replies back. On the other hand, your business logic is how your program works with data. It should be kept in the models or services, not the views. This makes the code easier to work with by making it more flexible.

# views.py  
from django.shortcuts import render  
from .models import BlogPost  
  
def blog\_post(request, blogpost\_id):  
 blogpost = BlogPost.objects.get(id=blogpost\_id)  
 return render(request, 'blogpost.html', {'blogpost': blogpost})  
  
# models.py  
from django.db import models  
  
class BlogPost(models.Model):  
 title = models.CharField(max\_length=200)  
 content = models.TextField()  
  
 def publish(self):  
 self.published\_date = timezone.now()  
 self.save()

2. **Use Class-Based Views for Reusable Components:**Compared to function-based views, class-based views may offer more reusable components. For simple tasks, however, function-based views might be a better choice.

Class-Based View  
from django.views import View  
  
class MyView(View):  
 def get(self, request):  
 # <view logic>  
 return HttpResponse('result')  
  
Function-Based View  
from django.http import HttpResponse  
  
def my\_view(request):  
 # <view logic>  
 return HttpResponse('result')

3. **Avoid Using Too Many Mixins:**Multiple inheritance is a type of mixin. They are a method for developing classes that can be applied in a variety of different contexts. However, using an excessive amount of mixins can make your code difficult to read and maintain.

# Too many mixins  
class MyView(TemplateView, ListView, DetailView, CreateView):  
 ...  
  
Simpler view  
class MyView(TemplateView):  
 ...

4. **Always Handle Exceptions:**If something goes wrong while a view is being processed, Django will automatically send back a 500 error. But you might want to make the error report easier to understand or do something else when an exception happens. This can be done by noticing when your ideas are wrong.

def my\_view(request):  
 try:  
 do\_something()  
 except SomeError as e:  
 return HttpResponse('An error occurred: %s' % e)

5. **Keep Your Views DRY (Don’t Repeat Yourself):** is a software development concept that cuts down on the amount of knowledge that is repeated in any way. If you write the same code in more than one view, you might want to turn that code into a different function or method.

Duplicated code  
def view1(request):  
 do\_something()  
 return HttpResponse('result')  
  
def view2(request):  
 do\_something()  
 return HttpResponse('result')  
  
# DRY code  
def do\_something():  
 # ...  
  
def view1(request):  
 do\_something()  
 return HttpResponse('result')  
  
def view2(request):  
 do\_something()  
 return HttpResponse('result')

6. **Use Django’s Built-In Decorators:**Django provides many built-in decorators, like @login\_required, @staff\_member\_required, etc. Use these decorators to easily handle common situations.

from django.contrib.auth.decorators import login\_required  
  
@login\_required  
def my\_view(request):  
 ...

7. **Use Django’s Generic Views:** Django comes with a number of basic views that can be used for common jobs in web development. Using these can speed up the creation process by a lot.

from django.views.generic import ListView  
from myapp.models import Post  
  
class PostListView(ListView):  
 model = Post

**Conclusion**

Views in Django are an essential aspect of every Django program. They act as channels for information to go from the user to the system and back. To fully appreciate the DRY (Don’t Repeat Yourself) concept at the center of the Django framework, you must be familiar with both Function-Based Views and Class-Based Views. As we’ve seen, the various points of view each have their advantages and niche applications. The straightforwardness and ease of use of Function-Based Views make them ideal for dealing with less complex tasks. Class-Based Views, on the other hand, provide an object-oriented strategy that works well in more complicated situations where reuse and extendability are essential. In the end, however, your project requirements should dictate whether you choose FBV or CBV. You’ll get experience creating Django apps and learn when each view type is most appropriate. The best way to learn is by doing, so keep at it. I hope your Django adventure goes swimmingly!

[Django](https://medium.com/tag/django?source=post_page-----4aab0e8d289c---------------django-----------------)

[Django Rest Framework](https://medium.com/tag/django-rest-framework?source=post_page-----4aab0e8d289c---------------django_rest_framework-----------------)

[Django Framework](https://medium.com/tag/django-framework?source=post_page-----4aab0e8d289c---------------django_framework-----------------)

[Backend Development](https://medium.com/tag/backend-development?source=post_page-----4aab0e8d289c---------------backend_development-----------------)

[Backend](https://medium.com/tag/backend?source=post_page-----4aab0e8d289c---------------backend-----------------)