# The Right Way



## The Pattern

This is how you start writing any HTML-based view in Django:

```
from django.template.response import TemplateResponse

def example_view(request, arg):
    return TemplateResponse(request, 'example.html', {})
```

And the corresponding urls.py:

```
from django.urls import path

from . import views

urlpatterns = [
    path('example/<str:arg>/', views.example_view, name='example_name'),
]
```

Which bits do you change?

- `example_view` should be a function name that describes your view e.g. `home` or `article_list`.
- `example.html` should be the path to the template you are using. I'm not going to cover how to write templates anywhere in this guide. Please see the Django tutorial on views and templates, and the Django template language topic.
- `{}`, the third argument to `TemplateResponse`, is the context data you want available in your template.
- `arg` is a placeholder for any number of optional URL parameters — parts of the URL path that you are matching with a path converter (here we used `str`) and supplying to the view function as a parameter. You can remove it, or add more, but have to change the URLconf to match.
- In `urls.py`, you change the arguments to `path` to be, respectively:
    - the matched URL (with any captured parts),
    - your view function defined above,
    - and an optional name that needs to be unique across your project, e.g. `home` or `myapp_articles_list`, to enable URL reversing.

That's it!

But, you need a slightly deeper understanding if you're going to write good Django views, so this page is quite a bit longer than most, for an explanation.

## The Explanation

First, it's vital to know what a view **is**. As the Django docs state:

> A view...is a Python function that takes a Web request and returns a Web response

Given that definition, what does your most basic view function look like? Time for Hello World! In your views.py, you'd have this:

```
from django.http import HttpResponse

def hello_world(request):
    return HttpResponse('Hello world!')
```

This function expects to receive a 'request' object as its parameter. This will be an instance of HttpRequest, which contains all the information about the request that the user's browser sent. It then returns an HttpResponse object as its return value. This contains all the data to be sent back to the user's browser — HTTP headers and body, which is typically a web page. In our case, we sent just the text `Hello world!`. This request-response cycle is the heart of the Django web framework.

In order to get Django to actually call our view function, we have to hook it into a "URLconf" somewhere. This is covered in the Django tutorial part 1, so I won't cover all the app layout stuff in detail — in brief, we'll have this in our urls.py:

```
from django.urls import path

from . import views

urlpatterns = [
    path('hello/', views.hello_world, name='hello_world'),
]
```

In many cases, we want a single view function to actually match a family of URLs which have some kind of parameter in them, and access that parameter in our view function. Django has built-in support for this. Suppose we want to match URLs like `/hello/XXX/` where `XXX` could be any string. Then our URLconf becomes:

```
urlpatterns = [
    path('hello/<str:my_arg>/', views.hello_world, name='hello_world'),
]
```

and our view signature:

```
def hello_world(request, my_arg):
    # etc
```

Now, for our classic web app, we are normally serving HTML i.e. web pages. Further, our HTML normally has bits we want to insert into it — this is a dynamic web site, not a static one — and we want to build it up in an ordered way that will handle HTML escaping, and also provide a common set of page elements (like navigation) for our different pages. So we'll almost always want to use Django's template engine — covered in the Django tutorial part 3. Instead of passing that `"Hello world"` string, we're going to have a `hello_world.html` template, and pass it some "context data" — any dynamic information that needs to appear in the page.

So our revised view might look like this:

```
from django.http import HttpResponse
from django.template import loader


def hello_world(request, my_arg):
    template = loader.get_template('hello_world.html')
    context = {}
    return HttpResponse(template.render(context, request))
```

Note that a template is not an essential part of a Django view — HTTP requests and responses are the essential parts. Templates are just a way of building up the body of the response. But for this kind of app, they are extremely common. So, as the Django tutorial notes, there is a shortcut for this process of loading a template, rendering it and putting it into a response — render(). With that, we can condense our view to this:

```
from django.shortcuts import render


def hello_world(request, my_arg):
    return render(request, 'hello_world.html', {})
```

The third parameter here is the empty context dictionary.

This is a great pattern for writing views. Django has one more trick up its sleeve, however — TemplateResponse.

The issue with just using `render` is that you get a plain `HttpResponse` object back that has no memory that it ever came from a template. Sometimes, however, it is useful to have functions return a value that does remember what it's "made of" — something that stores the template it is from, and the context. This can be

really useful in testing, but also if we want to something outside of our view function (such as decorators or middleware) to check or even change what's in the response before it finally gets 'rendered' and sent to the user.

For now, you can just accept that `TemplateResponse` is a more useful return value than a plain `HttpResponse`. (If you are already using `render` everywhere, there is absolutely no need to go and change it though, and almost everything in this guide will work exactly the same with `render` instead of `TemplateResponse`).

With that substitution, we've arrived at the pattern you'll want to start with for views:

```
from django.template.response import TemplateResponse

def example_view(request, arg):
    return TemplateResponse(request, 'example.html', {})
```

You need to know what each bit is, as described above. **But that is the end of the lesson**. You can skip to the next part. Or you can even just stop reading — you now know all the essentials of writing HTML views in Django.

You don't need to learn any of the CBV APIs - `TemplateView`, `ListView`, `DetailView`, `FormView`, `MultipleObjectMixin` etc. etc. and all their inheritance trees or method flowcharts. They will only make your life harder. Print out their documentation, put in a shed — or rather, a warehouse given how much there is — fill the warehouse with dynamite and don't look back.

Next up: How to do anything in a view.

## Discussion — keep the view viewable!

The most important thing about the pattern I'm recommending is that the view itself is visible. The fundamental nature of a view is that it is a function (or callable) that takes a request and returns a response. This is pretty obvious in the pattern above — we have:

- a function (check)
- that takes an argument called `request` (check)
- and returns some kind of response — a `TemplateResponse` (check)

Using CBVs we could rewrite the code as:

```
from django.views.generic import TemplateView

class ExampleView(TemplateView):
    template_name = "example.html"
```

But now **all 3 essential elements of the view have disappeared**. Where is the function or callable? Where is the request? Where is the response? To have any idea that this matches the description of what a view is you have to know what the base classes do, and the fundamental simplicity of what you are doing has been obscured.

There may be reasons for doing this, of course, but let's be aware of this problem and weigh up the advantages and disadvantages.

You might think "this is shorter than the FBV" is one of the advantages. It is, slightly, but as soon as you add the need for context data this advantage disappears, and we'll find we have more boilerplate, not less, with CBVs.