Writing views

A view function, or *view* for short, is a Python function that takes a web request and returns a web response. This response can be the HTML contents of a web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it's on your Python path. There's no other requirement–no "magic," so to speak. For the sake of putting the code *somewhere*, the convention is to put views in a file called `views.py`, placed in your project or application directory.

A simple view

Here's a view that returns the current date and time, as an HTML document:

```
from django.http import HttpResponse
import datetime


def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Let's step through this code one line at a time:

- First, we import the class `HttpResponse` from the `django.http` module, along with Python's `datetime` library.
- Next, we define a function called `current_datetime`. This is the view function. Each view function takes an `HttpRequest` object as its first parameter, which is typically named `request`.

  Note that the name of the view function doesn't matter; it doesn't have to be named in a certain way in order for Django to recognize it. We're calling it `current_datetime` here, because that name clearly indicates what it does.
- The view returns an `HttpResponse` object that contains the generated response. Each view function is responsible for returning an `HttpResponse` object. (There are exceptions, but we'll get to those later.)

Django's Time Zone: Django includes a `TIME_ZONE` setting that defaults to `America/Chicago`. This probably isn't where you live, so you might want to change it in your settings file.

Mapping URLs to views

So, to recap, this view function returns an HTML page that includes the current date and time. To display this view at a particular URL, you'll need to create a *URLconf*; see URL dispatcher for instructions.

Returning errors

Django provides help for returning HTTP error codes. There are subclasses of `HttpResponse` for a number of common HTTP status codes other than 200 (which means *"OK"*). You can find the full list of available subclasses in the request/response documentation. Return an instance of one of those subclasses instead of a normal `HttpResponse` in order to signify an error. For example:

```
from django.http import HttpResponse, HttpResponseNotFound


def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound("<h1>Page not found</h1>")
    else:
        return HttpResponse("<h1>Page was found</h1>")
```

There isn't a specialized subclass for every possible HTTP response code, since many of them aren't going to be that common. However, as documented in the `HttpResponse` documentation, you can also pass the HTTP status code into the constructor for `HttpResponse` to create a return class for any status code you like. For example:

```
from django.http import HttpResponse


def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

Because 404 errors are by far the most common HTTP error, there's an easier way to handle those errors.

The `Http404` exception

```
class django.http.Http404
```

When you return an error such as `HttpResponseNotFound`, you're responsible for defining the HTML of the resulting error page:

```
return HttpResponseNotFound("<h1>Page not found</h1>")
```

For convenience, and because it's a good idea to have a consistent 404 error page across your site, Django provides an `Http404` exception. If you raise `Http404` at any point in a view function, Django will catch it and return the standard error page for your application, along with an HTTP error code 404.

Example usage:

```
from django.http import Http404
from django.shortcuts import render
from polls.models import Poll


def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render(request, "polls/detail.html", {"poll": p})
```

In order to show customized HTML when Django returns a 404, you can create an HTML template named `404.html` and place it in the top level of your template tree. This template will then be served when `DEBUG` is set to `False`.

When `DEBUG` is `True`, you can provide a message to `Http404` and it will appear in the standard 404 debug template. Use these messages for debugging purposes; they generally aren't suitable for use in a production 404 template.

Customizing error views

The default error views in Django should suffice for most web applications, but can easily be overridden if you need any custom behavior. Specify the handlers as seen below in your URLconf (setting them anywhere else will have no effect).

The `page_not_found()` view is overridden by `handler404`:

```
handler404 = "mysite.views.my_custom_page_not_found_view"
```

The `server_error()` view is overridden by `handler500`:

```
handler500 = "mysite.views.my_custom_error_view"
```

The `permission_denied()` view is overridden by `handler403`:

```
handler403 = "mysite.views.my_custom_permission_denied_view"
```

The `bad_request()` view is overridden by `handler400`:

```
handler400 = "mysite.views.my_custom_bad_request_view"
```

See also: Use the `CSRF_FAILURE_VIEW` setting to override the CSRF error view.

Testing custom error views

To test the response of a custom error handler, raise the appropriate exception in a test view. For example:

```
from django.core.exceptions import PermissionDenied
from django.http import HttpResponse
from django.test import SimpleTestCase, override_settings
from django.urls import path


def response_error_handler(request, exception=None):
    return HttpResponse("Error handler content", status=403)


def permission_denied_view(request):
    raise PermissionDenied


urlpatterns = [
    path("403/", permission_denied_view),
]

handler403 = response_error_handler


# ROOT_URLCONF must specify the module that contains handler403 = ...
@override_settings(ROOT_URLCONF=__name__)
class CustomErrorHandlerTests(SimpleTestCase):
    def test_handler_renders_template_response(self):
        response = self.client.get("/403/")
        # Make assertions on the response here. For example:
        self.assertContains(response, "Error handler content", status_code=403)
```

Async views

As well as being synchronous functions, views can also be asynchronous (“async”) functions, normally defined using Python’s `async def` syntax. Django will automatically detect these and run them in an async context. However, you will need to use an async server based on ASGI to get their performance benefits.

Here’s an example of an async view:

```
import datetime
from django.http import HttpResponse


async def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

You can read more about Django's async support, and how to best use async views, in Asynchronous support.