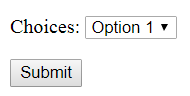
class SearchForm(forms.Form):

c = [("1", "Option 1"), ("2", "Option 2")]

choices = forms.ChoiceField(choices=c, label="Choices")



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**What is a context?**

When you use a Django Template, it is compiled once (and only once) and stored for future use, as an optimization. A template can have variable names in double curly braces, such as {{ myvar1 }}, {{ myvar2 }}.

A Context is a dictionary with variable names as the "key" and their values as the "value". Hence if your context for the above template looks like: {myvar1: 101, myvar2: 102}, when you pass this context to the template render method, {{ myvar1 }} would be replaced with 101 and {{ myvar2 }} with 102 in your template. This is a simplistic example, but really a Context object is the "Context" in which the template is being rendered.

Or, A **context** is a **variable name** => **variable value** mapping that is passed to a template.

**Context processors** let you specify a number of variables that get set in each **context** automatically – without you having to specify the variables in each **render()** call.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Django.views.generic

In some cases, writing views, as we have seen earlier is really heavy. Imagine you need a static page or a listing page. Django offers an easy way to set those simple views that is called generic views.

Unlike classic views, generic views are classes not functions. Django offers a set of classes for generic views in django.views.generic, and every generic view is one of those classes or a class that inherits from one of them.

There are 10+ generic classes −

>>> import django.views.generic

>>> dir(django.views.generic)

['ArchiveIndexView', 'CreateView', 'DateDetailView', 'DayArchiveView',

'DeleteView', 'DetailView', 'FormView', 'GenericViewError', 'ListView',

'MonthArchiveView', 'RedirectView', 'TemplateView', 'TodayArchiveView',

'UpdateView', 'View', 'WeekArchiveView', 'YearArchiveView', '\_\_builtins\_\_',

'\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_path\_\_', 'base', 'dates',

'detail', 'edit', 'list']

This you can use for your generic view. Let's look at some example to see how it works.

Static Pages

Let's publish a static page from the “static.html” template.

Our static.html −

<html>

<body>

This is a static page!!!

</body>

</html>

If we did that the way we learned before, we would have to change the **myapp/views.py** to be −

from django.shortcuts import render

def static(request):

return render(request, 'static.html', {})

and **myapp/urls.py** to be −

from django.conf.urls import patterns, url

urlpatterns = patterns("myapp.views", url(r'^static/', 'static', name = 'static'),)

The best way is to use generic views. For that, our myapp/views.py will become −

from django.views.generic import TemplateView

class StaticView(TemplateView):

template\_name = "static.html"

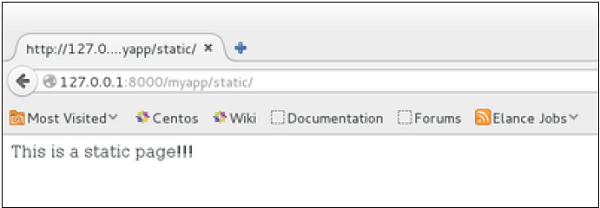
And our myapp/urls.py we will be −

from myapp.views import StaticView

from django.conf.urls import patterns

urlpatterns = patterns("myapp.views", (r'^static/$', StaticView.as\_view()),)

When accessing /myapp/static you get −



For the same result we can also, do the following −

* No change in the views.py
* Change the url.py file to be −

from django.views.generic import TemplateView

from django.conf.urls import patterns, url

urlpatterns = patterns("myapp.views",

url(r'^static/',TemplateView.as\_view(template\_name = 'static.html')),)

As you can see, you just need to change the url.py file in the second method.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Django is SECURED!: By default, it prevents most common security threats.

* XSS (cross-site scripting) protection — Django template system by default escapes variables, unless they are explicitly marked as safe.
* CSRF (cross site request forgery) protection — easy to turn on globally, guarantees that forms (POST requests) are sent from your own site.
* SQL injection protection — Django uses built-in ORM, thus there is no risk of SQL injection (raw queries are possible, but by no means something that a beginner would need to use).

     Additional security features:

* [Clickjacking protection](https://docs.djangoproject.com/en/stable/ref/clickjacking/) — Django can detect when the content is requested from unauthorized iframe
* Safe password hash — Django by default uses [PBKDF2](http://en.wikipedia.org/wiki/PBKDF2), another option is [bcrypt](http://en.wikipedia.org/wiki/Bcrypt). Both are resilient to usage of rainbow tables (thanks to salt), both have significant compute time to prevent easy bruteforce.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

related\_name

What is the related\_name argument useful for on ManyToManyField and ForeignKey fields? For example, given the following code, what is the effect of related\_name='maps'?

class Map(db.Model):

members = models.ManyToManyField(User, related\_name='maps',

verbose\_name=\_('members'))

The related\_name attribute specifies the name of the reverse relation from the User model back to your model.

If you don't specify a related\_name, Django automatically creates one using the name of your model with the suffix \_set, for instance User.map\_set.all().

If you *do* specify, e.g. related\_name=maps on the User model, User.map\_set will still work, but the User.maps. syntax is obviously a bit cleaner and less clunky; so for example, if you had a user object current\_user, you could use current\_user.maps.all() to get all instances of your Mapmodel that have a relation to current\_user.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[What is actually assertEquals in Python?](https://stackoverflow.com/questions/17920625/what-is-actually-assertequals-in-python)

I have the following test.py file in django. can you please explain this code?

from contacts.models import Contact

...

class ContactTests(TestCase):

"""Contact model tests."""

def test\_str(self):

contact = Contact(first\_name='John', last\_name='Smith')

self.assertEquals(

str(contact),

'John Smith',

)

Answer:

Basically it will check if str(contact) == 'John Smith', if not then assert equal is failed and the test is failed and it will notify you the error at that line.

In other words, assertEquals is a function to check if two variables are equal, for purposes of automated testing:

def assertEquals(var1, var2):

if var1 == var2:

return True

else:

return False

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

get\_queryset()

Get the list of items for this view. This must be an iterable and may be a queryset (in which queryset-specific behavior will be enabled).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_