**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.

All views inherit from the [**View**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.View) class, which handles linking the view in to the URLs, HTTP method dispatching and other simple features.

[**RedirectView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.RedirectView) is for a simple HTTP redirect, and [**TemplateView**](https://docs.djangoproject.com/en/2.1/ref/class-based-views/base/#django.views.generic.base.TemplateView) extends the base class to make it also render a template.

**Introduction to class-based views**

Class-based views provide an alternative way to implement views as Python objects instead of functions.

They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

* Organization of code related to specific HTTP methods (**GET**, **POST**, etc.) can be addressed by separate methods instead of conditional branching.
* Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

**Steps to Create a class based view**

* To create a class-based view, start by creating a class that inherits from **django.views.generic.View** or one of its subclasses.
* In your path, specify the view method as the name of the new class, plus **.as\_view().**

**Example:**

path(**''**, Sample.as\_view()),  
path(**'open/'**, Sample.as\_view()),

* In your class, write a get method that takes as arguments self (as always), request (the HttpRequest), and any other arguments from the request as specified in your URLconf.
* In your get method, use the same logic you'd have used in an old view, except that you can assume the request method is GET. Return an HttpResponse as usual.
* If you need to handle POST, write a post method, just like your get method except that you can assume the request method is POST.
* Any request method that you don't write a handler method for will automatically get back a "method not allowed" response; you don't have to do anything special.

**Example**

**from** **django.http** **import** HttpResponse

**from** **django.views** **import** View

**class** **MyView**(View):

**def** get(self, request):

*# <view logic>*

**return** HttpResponse('result')

***# urls.py***

**from** **django.urls** **import** path

**from** **myapp.views** **import** MyView

urlpatterns = [

path('about/', MyView.as\_view()),

]

**Built-in class-based views**

* **Base views**
  + View
  + TemplateView
  + RedirectView
* **Generic display views**
  + DetailView
  + ListView
* **Generic editing views**
  + FormView
  + CreateView
  + UpdateView
  + DeleteView

The available views that you’ll probably use the most:

* **TemplateView** – render a given template
* **RedirectView** – redirect to a given URL
* **DetailView** – show full details of an object
* **ListView** – show a collection of an object
* **CreateView** – render a form to create an object, provides validation and updates database
* **UpdateView** – render a form to edit an object, provides validation and updates database
* **DeleteVew** – GET method show confirmation screen, POST method delete object from database

**Base views**

Many of Django’s built-in class-based views inherit from other class-based views or various mixins

**View**

***class*django.views.generic.base.View**

The master class-based base view. All other class-based views inherit from this base class. It isn’t strictly a generic view and thus can also be imported from django.views.

**TemplateView**

***class*django.views.generic.base.TemplateView**

TemplateView **should be used** when you want to present some information in a html page.

TemplateView **shouldn’t be used** when your page has forms and does creation or update of objects.

TemplateView is most suitable in following cases:

* Showing ‘about us’ like pages which are static and hardly needs any context. Though it is easy to use context variables with TemplateView.
* Showing pages which work with GET requests and don’t have forms in it.

**Example**

**from** django.contrib **import** admin  
**from** django.urls **import** path  
**from** django.views.generic **import** TemplateView  
  
urlpatterns = [  
 path(**'admin/'**, admin.site.urls),  
 path(**''**,TemplateView.as\_view(template\_name=**"aboutus.html"**))  
]

**Note:** Create **"aboutus.html"** in templates folder.

**Django Redirects**

In Django, you redirect the user to another URL by returning an instance of HttpResponseRedirect or HttpResponsePermanentRedirect from your view.

**The RedirectView Function Based View**

The simplest way to do this is to use the function [redirect()](https://docs.djangoproject.com/en/2.1/topics/http/shortcuts/#redirect) from the module **django.shortcuts.dd**

**def registerUser(request):**

**res = redirect("/index/")  
 return res**

**An absolute or relative URL.**

from django.shortcuts import redirect

def relative\_url\_view(request):

return redirect('/posts/archive/')

# equivalent to: return HttpResponseRedirect('/posts/archive/')

def absolute\_url\_view(request):

return redirect('http://www.facebook.com')

**The RedirectView Class-Based View**

**If you have a view that does nothing but returning a redirect, you could use the class-based view django.views.generic.base.RedirectView.**

**Example ( in urls.py )**

from django.views.generic import RedirectView  
urlpatterns = [  
 path('opengoogle/',RedirectView.as\_view(url="http://www.google.com"))  
]

**Example ( in views.py using a class )**

**views.py**

**class MyFacebook(RedirectView):  
 url = "https://facebook.com"**

**urls.py**

**from apprv.views import MyFacebook  
urlpatterns = [  
 path('openfacebook/',MyFacebook.as\_view())  
]**

**Passing Parameters with Redirects**

The option is to pass the data in the query string of your redirect URL, which means redirecting like:

**Example**

http://example.com/redirect-path/?parameter=value

**Generic display views**

**1. DetailView**

DetailView should be used when you want to present detail of a single model instance.

DetailView shouldn’t be used when your page has forms and does creation or update of objects

**2. ListView**

Django provides several class based generic views to accomplish common tasks. One among them is ListView.

ListView should be used when you want to present a list of objects in a html page.

ListView shouldn’t be used when your page has forms and does creation or update of objects.

Note: - TemplateView can achieve everything which ListView can, but ListView avoids a lot of boilerplate code which would be needed with TemplateView.

**Boilerplate code**

In computer programming, **boilerplate code** or **boilerplate** refers to sections of code that have to be included in many places with little or no alteration. It is often used when referring to languages that are considered *verbose*, i.e. the programmer must write a lot of code to do minimal jobs.

**views.py**

from .models import PersonInfo

from django.views.generic import ListView

class ViewAllDetails(ListView):

queryset = PersonInfo.objects.all()

**Urls.py**

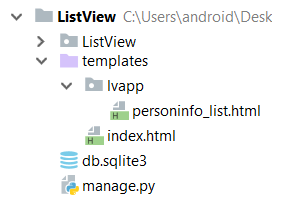
**from** lvapp.views **import** SaveDetails,ViewAllDetails

urlpatterns = [

path(**'viewdetails/'**, ViewAllDetails.as\_view()),]

**Create html(template) file**

DIR Structure :



**Example**

<body>

{% for x in object\_list %}

{{ x.cno }} ---

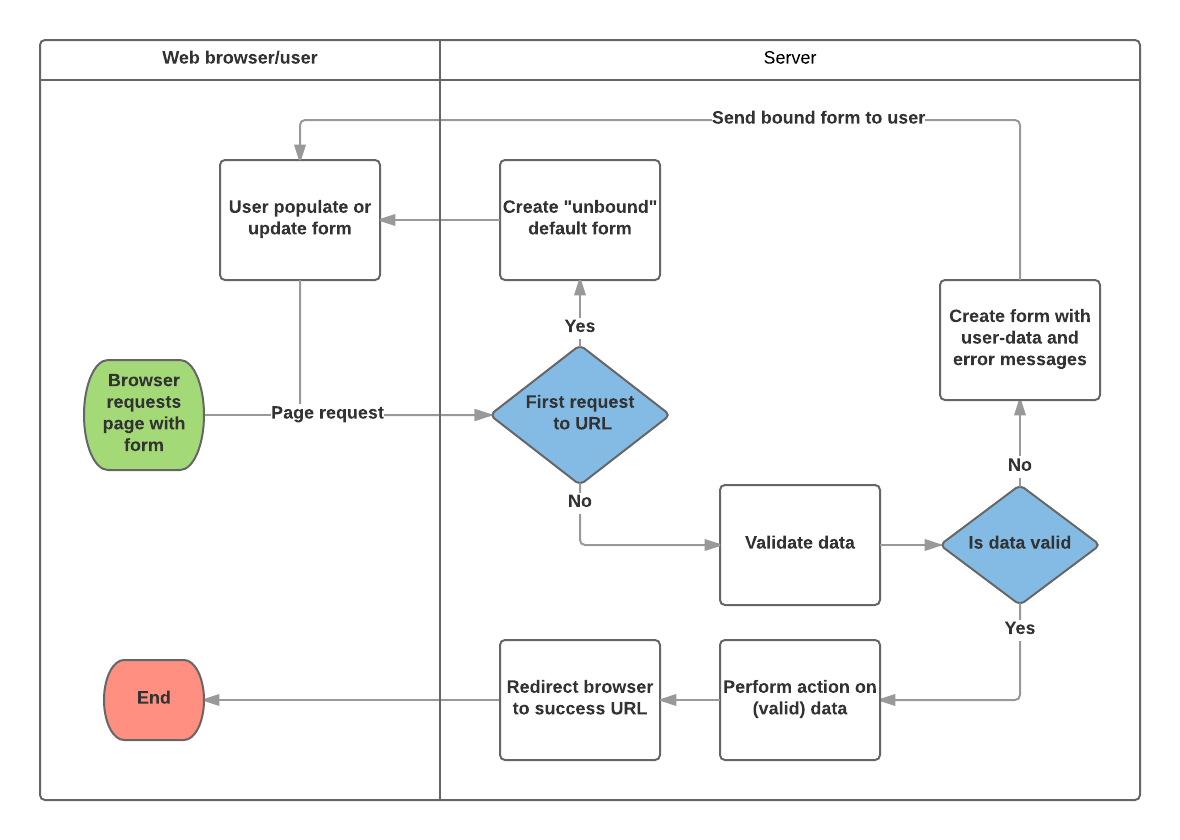
{{ x.pname }} <br>

{% endfor %}

</body>

**Generic editing views**

**1. FormView**



Based on the diagram above, the main things that Django's form handling does are:

1. Display the default form the first time it is requested by the user.
   * The form may contain blank fields (e.g. if you're creating a new record), or it may be pre-populated with initial values (e.g. if you are changing a record, or have useful default initial values).
   * The form is referred to as *unbound* at this point, because it isn't associated with any user-entered data (though it may have initial values).
2. Receive data from a submit request and bind it to the form.
   * Binding data to the form means that the user-entered data and any errors are available when we need to redisplay the form.
3. Clean and validate the data.
   * Cleaning the data performs sanitization of the input (e.g. removing invalid characters that might be used to send malicious content to the server) and converts them into consistent Python types.
   * Validation checks that the values are appropriate for the field (e.g. are in the right date range, aren't too short or too long, etc.)
4. If any data is invalid, re-display the form, this time with any user populated values and error messages for the problem fields.
5. If all data is valid, perform required actions (e.g. save the data, send and email, return the result of a search, upload a file etc.)
6. Once all actions are complete, redirect the user to another page.

**Form rendering options**

Don’t forget that a form’s output does *not* include the surrounding <form> tags, or the form’s submit control. You will have to provide these yourself.

* {{ form.as\_table }} will render them as table cells wrapped in <tr> tags
* {{ form.as\_p }} will render them wrapped in <p> tags
* {{ form.as\_ul }} will render them wrapped in <li> tags

Note that you’ll have to provide the surrounding <table> or <ul>

elements yourself.

**Rendering fields manually**

We don’t have to let Django unpack the form’s fields; we can do it manually if we like. Each field is available as an attribute of the form using **{{form.name\_of\_field }},** and in a Django template, will be rendered appropriately.

**Example to display Field Label, Field and Help Text**

<div class="alert alert-danger" role="alert">  
 {{ form.idno.label\_tag }} {{ form.idno }} {{ form.idno.help\_text }}  
</div>  
<div class="alert alert-danger" role="alert">  
 {{ form.name.label\_tag }}{{ form.name}} {{ form.name.help\_text }}  
</div>  
<div class="alert alert-danger" role="alert">  
 {{ form.salary.label\_tag }}{{ form.salary }}{{ form.salary.help\_text }}  
</div>

**Rendering form error messages**

To display Error messages we use **{{form.non\_field\_errors }}** at the top of the form and the template lookup for errors on each field.

Using **{{ form.name\_of\_field.errors }}** displays a list of form errors, rendered as an unordered list.

**Looping over the form’s fields**

If you’re using the same HTML for each of your form fields, you can reduce duplicate code by looping through each field in turn using a **{%for %}** loop:

{% for field in form %}  
 <div>  
 {{ field.errors }}  
 {{ field.label\_tag }} {{ field }}  
 {% if field.help\_text %}  
 <p> {{ field.help\_text|safe }}</p>  
 {% endif %}  
 </div>  
{% endfor %}

**Widgets**

A widget is Django’s representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.

**Example**

class CommentForm(forms.Form):

name = forms.CharField()

url = forms.URLField()

comment = forms.CharField(widget=forms.Textarea)

This would specify a form with a comment that uses a larger [Textarea](https://docs.djangoproject.com/en/2.1/ref/forms/widgets/#django.forms.Textarea) widget, rather than the default [TextInput](https://docs.djangoproject.com/en/2.1/ref/forms/widgets/#django.forms.TextInput) widget.

**Providing Attributes to Widget's (attrs)**

A dictionary containing HTML attributes to be set on the rendered widget.

from django import forms

forms.TextInput(attrs={'size': 10, 'title': 'Your name'})

**Output :** <input title="Your name" type="text" name="name" value="A name" size="10">

**Built-in widgets**

TextInput : input\_type: 'text'

NumberInput : input\_type: 'number'

EmailInput : input\_type: 'email'

URLInput : input\_type: 'url'

PasswordInput : input\_type: 'password'

HiddenInput : input\_type: 'hidden'

DateInput : input\_type: 'text'

format : DATE\_INPUT\_FORMATS

DateTimeInput : input\_type: 'text'

format : DATETIME\_INPUT\_FORMATS

TimeInput : input\_type: 'text'

format : TIME\_INPUT\_FORMATS

Textarea : Renders as: <textarea>...</textarea>

**Selector and checkbox widgets**

**These widgets make use of the HTML elements**

**<select>,**

**<input type="checkbox">, and**

**<input type="radio">.**

Widgets that render multiple choices have anoption\_template\_name attribute that specifies the template used to render each choice.

For example, for the Select widget, select\_option.html renders the **<option> for a <select>.**

CheckboxInput : input\_type: 'checkbox'

Select : Renders as: <select><option ...>...</select>

SelectMultiple : Similar to Select, but allows multiple

selection: <select multiple>...</select>

RadioSelect : Similar to Select, but rendered as a list of radio buttons within <li> tags:

Example

<ul>

<li><input type="radio" name="..."></li>

</ul>

**Note:** To render you can loop over the radio buttons in the template. Assuming a form myform with a field beatles that uses a RadioSelect as its widget:

{% for radio in myform.beatles %}

<div class="myradio">

{{ radio }}

</div>

{% endfor %}

File upload widgets : Renders as: <input type="file" ...>

**2. CreateView**

**3. UpdateView**

**4. DeleteView**