Type of Views

- Function Based View
- Class Based View

Class Based View

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

They do not replace function-based views.

- Base Class-Based Views / Base View
- Generic Class-Based Views / Generic View

Advantages:-

- 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.

Base Class-Based View

Base class-based views can be thought of as parent views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins which extend what base views can do.

- View
- TemplateView
- RedirectView

Generic Class Based View

Django's generic views are built off of those base views, and were developed as a shortcut for common usage patterns such as displaying the details of an object.

They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to repeat yourself.

Most generic views require the queryset key, which is a QuerySet instance.

- Display View – ListView, DetailView
- Editing View – FormView, CreateView, UpdateView, DeleteView
- Date Views – ArchiveIndexView, YearArchiveView, MonthArchiveView, WeekArchiveView, DayArchiveView, TodayArchiveView, DateDetailView

Generic Display View

The two following generic class-based views are designed to display data.

- ListView
- DetailView

ListView

django.views.generic.list.ListView

A page representing a list of objects.

While this view is executing, self.object_list will contain the list of objects (usually, but not necessarily a queryset) that the view is operating upon.

This view inherits methods and attributes from the following views:

- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.list.BaseListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.base.View

MultipleObjectTemplateResponseMixin

A mixin class that performs template-based response rendering for views that operate upon a list of object instances. Requires that the view it is mixed with provides self.object_list, the list of object instances that the view is operating on. self.object_list may be, but is not required to be, a QuerySet.

This inherits methods and attributes from the following views:

- django.views.generic.base.TemplateResponseMixin

Attribute:-

template_name_suffix - The suffix to append to the auto-generated candidate template name. Default suffix is _list.

Method:-

get_template_names() - It returns a list of candidate template names.

BaseListView

A base view for displaying a list of objects. It is not intended to be used directly, but rather as a parent class of the django.views.generic.list.ListView or other views representing lists of objects.

This view inherits methods and attributes from the following views:

- django.views.generic.list.MultipleObjectMixin
- django.views.generic.base.View

Methods:-

get(request, *args, **kwargs) - It adds object_list to the context. If allow_empty is True then display an empty list. If allow_empty is False then raise a 404 error.

MultipleObjectMixin

django.views.generic.list.MultipleObjectMixin

A mixin that can be used to display a list of objects.

If paginate_by is specified, Django will paginate the results returned by this. You can specify the page number in the URL in one of two ways:

- Use the page parameter in the URLconf.
- Pass the page number via the page query-string parameter.

These values and lists are 1-based, not 0-based, so the first page would be represented as page 1.

As a special case, you are also permitted to use last as a value for page.

This allows you to access the final page of results without first having to determine how many pages there are.

Note that page must be either a valid page number or the value last; any other value for page will result in a 404 error.

MultipleObjectMixin

Attribute:-

allow_empty - A boolean specifying whether to display the page if no objects are available. If this is False and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is True.

model - The model that this view will display data for. Specifying model = Student is effectively the same as specifying queryset = Student.objects.all(), where objects stands for Student's default manager.

queryset - A QuerySet that represents the objects. If provided, the value of queryset supersedes the value provided for model.

ordering - A string or list of strings specifying the ordering to apply to the queryset. Valid values are the same as those for order_by().

MultipleObjectMixin

Attributes:-

paginate_by - An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with paginate_by objects per page. The view will expect either a page query string parameter (via request.GET) or a page variable specified in the URLconf.

paginate_orphans - An integer specifying the number of “overflow” objects the last page can contain. This extends the paginate_by limit on the last page by up to paginate_orphans, in order to keep the last page from having a very small number of objects.

page_kwarg - A string specifying the name to use for the page parameter. The view will expect this parameter to be available either as a query string parameter (via request.GET) or as a kwarg variable specified in the URLconf. Defaults to page.

MultipleObjectMixin

Attributes:-

paginator_class - The paginator class to be used for pagination. By default, django.core.paginator.Paginator is used. If the custom paginator class doesn’t have the same constructor interface as django.core.paginator.Paginator, you will also need to provide an implementation for get_paginator().

context_object_name - Designates the name of the variable to use in the context.

MultipleObjectMixin

Methods:-

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).

get_ordering() - Returns a string (or iterable of strings) that defines the ordering that will be applied to the queryset.

Returns ordering by default.

paginate_queryset(queryset, page_size) - Returns a 4-tuple containing (paginator, page, object_list, is_paginated).

Constructed by paginating queryset into pages of size page_size. If the request contains a page argument, either as a captured URL argument or as a GET argument, object_list will correspond to the objects from that page.

MultipleObjectMixin

Methods:-

get_paginate_by(queryset) - Returns the number of items to paginate by, or None for no pagination. By default this returns the value of paginate_by.

get_paginator(queryset, per_page, orphans=0, allow_empty_first_page=True) - Returns an instance of the paginator to use for this view. By default, instantiates an instance of paginator_class.

get_paginate_orphans() - An integer specifying the number of “overflow” objects the last page can contain. By default this returns the value of paginate_orphans.

get_allow_empty() - Return a boolean specifying whether to display the page if no objects are available. If this method returns False and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is True.

MultipleObjectMixin

Methods:-

get_context_object_name(object_list) - Return the context variable name that will be used to contain the list of data that this view is manipulating. If object_list is a queryset of Django objects and context_object_name is not set, the context name will be the model_name of the model that the queryset is composed from, with postfix '_list' appended. For example, the model Article would have a context object named article_list.

get_context_data(**kwargs) - Returns context data for displaying the list of objects.

Context

object_list: The list of objects that this view is displaying. If context_object_name is specified, that variable will also be set in the context, with the same value as object_list.

is_paginated: A boolean representing whether the results are paginated. Specifically, this is set to False if no page size has been specified, or if the available objects do not span multiple pages.

paginator: An instance of django.core.paginator.Paginator. If the page is not paginated, this context variable will be None.

page_obj: An instance of django.core.paginator.Page. If the page is not paginated, this context variable will be None.

ListView with Default Template and Context

views.py

```
from django.views.generic.list import ListView
from .models import Student
class StudentListView(ListView):
    model = Student
```

urls.py

```
urlpatterns = [  path('student/', views.StudentListView.as_view(), name='student'),  ]
```

Default Template

Syntax:- AppName/ModelClassName_list.html

Example:- school/student_list.html

Default Context

Syntax:- ModelClassName_list
Example:- student_list
We can also use **object_list**

ListView with Custom Template and Context

views.py

```
from django.views.generic.list import ListView
from .models import Student
class StudentListView(ListView):
    model = Student
    template_name = ‘school/student.html’
    context_object_name = ‘students’
```

Custom Template Name

Custom Context Name

urls.py

```
urlpatterns = [
    path('student/', views.StudentListView.as_view(), name='student’),
]
```

Note - *school/students.html*, *school/student_list.html* These both will work

DetailView

django.views.generic.detail.DetailView

While this view is executing, self.object will contain the object that the view is operating upon.

This view inherits methods and attributes from the following views:

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.detail.BaseDetailView
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.base.View

SingleObjectTemplateResponseMixin

django.views.generic.detail.SingleObjectTemplateResponseMixin

A mixin class that performs template-based response rendering for views that operate upon a single object instance. Requires that the view it is mixed with provides self.object, the object instance that the view is operating on. self.object will usually be, but is not required to be, an instance of a Django model. It may be None if the view is in the process of constructing a new instance.

This view inherits methods and attributes from the following views:

- django.views.generic.base.TemplateResponseMixin

SingleObjectTemplateResponseMixin

Attribute:-

template_name_field - The field on the current object instance that can be used to determine the name of a candidate template. If either template_name_field itself or the value of the template_name_field on the current object instance is None, the object will not be used for a candidate template name.

template_name_suffix - The suffix to append to the auto-generated candidate template name. Default suffix is _detail.

Method:-

get_template_names()- Returns a list of candidate template names. Returns the following list:

the value of template_name on the view (if provided)

the contents of the template_name_field field on the object instance that the view is operating upon (if available)

<app_label>/<model_name><template_name_suffix>.html

SingleObjectMixin

django.views.generic.detail.SingleObjectMixin

Provides a mechanism for looking up an object associated with the current HTTP request.

Attribute:-

model - The model that this view will display data for. Specifying model = Student is effectively the same as specifying queryset = Student.objects.all(), where objects stands for Student's default manager.

queryset - A QuerySet that represents the objects. If provided, the value of queryset supersedes the value provided for model.

slug_field - The name of the field on the model that contains the slug. By default, slug_field is 'slug'.

slug_url_kwarg - The name of the URLConf keyword argument that contains the slug. By default, slug_url_kwarg is 'slug'.

SingleObjectMixin

Attribute:-

pk_url_kwarg - The name of the URLConf keyword argument that contains the primary key. By default, pk_url_kwarg is 'pk'.

context_object_name - Designates the name of the variable to use in the context.

query_pk_and_slug - If True, causes get_object() to perform its lookup using both the primary key and the slug. Defaults to False.

SingleObjectMixin

Methods:-

get_object(queryset=None) - Returns the single object that this view will display. If queryset is provided, that queryset will be used as the source of objects; otherwise, get_queryset() will be used. get_object() looks for a pk_url_kwarg argument in the arguments to the view; if this argument is found, this method performs a primary-key based lookup using that value. If this argument is not found, it looks for a slug_url_kwarg argument, and performs a slug lookup using the slug_field.

When query_pk_and_slug is True, get_object() will perform its lookup using both the primary key and the slug.

get_queryset() - Returns the queryset that will be used to retrieve the object that this view will display. By default, get_queryset() returns the value of the queryset attribute if it is set, otherwise it constructs a QuerySet by calling the all() method on the model attribute's default manager.

get_slug_field() - Returns the name of a slug field to be used to look up by slug. By default this returns the value of slug_field.

SingleObjectMixin

Methods:-

get_context_object_name(obj) - Return the context variable name that will be used to contain the data that this view is manipulating. If context_object_name is not set, the context name will be constructed from the model_name of the model that the queryset is composed from. For example, the model Article would have context object named 'article'.

get_context_data(**kwargs) - Returns context data for displaying the object.

The base implementation of this method requires that the self.object attribute be set by the view (even if None). Be sure to do this if you are using this mixin without one of the built-in views that does so.

It returns a dictionary with these contents:

object: The object that this view is displaying (self.object).

context_object_name: self.object will also be stored under the name returned by get_context_object_name(), which defaults to the lowercased version of the model name.

DetailView with Default Template & Context

views.py

```
from django.views.generic.detail import DetailView
from .models import Student
class StudentDetailView(DetailView):
    model = Student
```

urls.py

```
urlpatterns = [    path('student/<int:pk>', views.StudentDetailView.as_view(), name='student'),    ]
```

Default Template

Syntax:- AppName/ModelClassName_detail.html

Example:- school/student_detail.html

Default Context

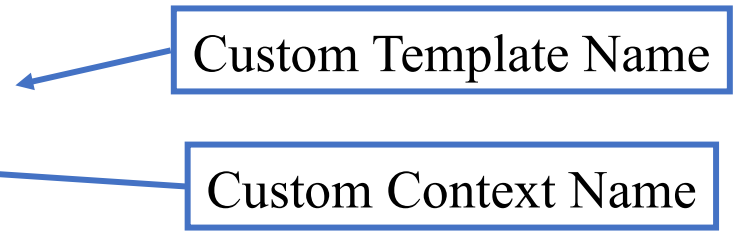
Syntax:- ModelClassName

Example:= student

DetailView with Custom Template & Context

views.py

```
from django.views.generic.detail import DetailView
from .models import Student
class StudentDetailView(DetailView):
        model = Student
        template_name = ‘school/student.html’
        context_object_name = ‘student’
```



urls.py

```
urlpatterns = [
        path('student/<int:pk>', views.StudentDetailView.as_view(), name='student’),
]
```

Generic Editing View

The following views are described on this page and provide a foundation for editing content:

- FormView
- CreateView
- UpdateView
- DeleteView

FormView

django.views.generic.edit.FormView

A view that displays a form. On error, redisplays the form with validation errors; on success, redirects to a new URL.

This view inherits methods and attributes from the following views:

- django.views.generic.base.TemplateResponseMixin
- django.views.generic.edit.BaseFormView
- django.views.generic.edit.FormMixin
- django.views.generic.edit.ProcessFormView
- django.views.generic.base.View

FormMixin

django.views.generic.edit.FormMixin

A mixin class that provides facilities for creating and displaying forms.

This view inherits methods and attributes from the following views:

- django.views.generic.base.ContextMixin

Attributes:-

initial - A dictionary containing initial data for the form.

form_class - The form class to instantiate.

success_url - The URL to redirect to when the form is successfully processed.

prefix - The prefix for the generated form.

FormMixin

Methods:-

get_initial() - Retrieve initial data for the form. By default, returns a copy of initial.

get_form_class() - Retrieve the form class to instantiate. By default form_class.

get_form(form_class=None) - Instantiate an instance of form_class using get_form_kwargs(). If form_class isn’t provided get_form_class() will be used.

get_form_kwargs() - Build the keyword arguments required to instantiate the form.

The initial argument is set to get_initial(). If the request is a POST or PUT, the request data (request.POST and request.FILES) will also be provided.

FormMixin

Methods:-

get_prefix() - Determine the prefix for the generated form. Returns prefix by default.

get_success_url() - Determine the URL to redirect to when the form is successfully validated. Returns success_url by default.

form_valid(form) - Redirects to get_success_url().

form_invalid(form) - Renders a response, providing the invalid form as context.

get_context_data(**kwargs) - Calls get_form() and adds the result to the context data with the name ‘form’.

ProcessFormView

django.views.generic.edit.ProcessFormView

A mixin that provides basic HTTP GET and POST workflow.

Extends

django.views.generic.base.View

Methods:-

get(request, *args, **kwargs) - Renders a response using a context created with get_context_data().

post(request, *args, **kwargs) - Constructs a form, checks the form for validity, and handles it accordingly.

put(*args, **kwargs) - The PUT action is also handled and passes all parameters through to post().

FormView

forms.py

```
from django import forms
class ContactForm(forms.Form):
 name = forms.CharField()
 email = forms.EmailField()
 msg = forms.CharField(widget=forms.Textarea)
```

urls.py

```
from school import views
urlpatterns = [
path('contact/', views.ContactFormView.as_view(),
name='contact'),
path('thankyou/', views.ThanksTemplateView.as_view(),
name='thankyou'),
]
```

views.py

```
from django.views.generic.edit import FormView
class ContactFormView(FormView):
 template_name = 'school/contact.html'
 form_class = ContactForm
 success_url = '/thankyou/'
 def form_valid(self, form):
        print(form)
        print(form.cleaned_data['name’])
        return HttpResponse('Msg Sent’)


class ThanksTemplateView(TemplateView):
 template_name = 'school/thankyou.html'
```

CreateView

django.views.generic.edit.CreateView

A view that displays a form for creating an object, redisplaying the form with validation errors (if there are any) and saving the object.

This view inherits methods and attributes from the following views:

django.views.generic.detail.SingleObjectTemplateResponseMixin

django.views.generic.base.TemplateResponseMixin

django.views.generic.edit.BaseCreateView

django.views.generic.edit.ModelFormMixin

django.views.generic.edit.FormMixin

django.views.generic.detail.SingleObjectMixin

django.views.generic.edit.ProcessFormView

django.views.generic.base.View

CreateView

Attributes:-

template_name_suffix - The CreateView page displayed to a GET request uses a template_name_suffix of '_form'.

object - When using CreateView you have access to self.object, which is the object being created. If the object hasn't been created yet, the value will be None.

ModelFormMixin

A form mixin that works on ModelForms, rather than a standalone form.

Since this is a subclass of SingleObjectMixin, instances of this mixin have access to the model and queryset attributes, describing the type of object that the ModelForm is manipulating.

If you specify both the fields and form_class attributes, an ImproperlyConfigured exception will be raised.

Mixins

django.views.generic.edit.FormMixin

django.views.generic.detail.SingleObjectMixin

ModelFormMixin

Attributes

model - A model class. Can be explicitly provided, otherwise will be determined by examining self.object or queryset.

fields - A list of names of fields. This is interpreted the same way as the Meta.fields attribute of ModelForm.

This is a required attribute if you are generating the form class automatically (e.g. using model). Omitting this attribute will result in an ImproperlyConfigured exception.

success_url - The URL to redirect to when the form is successfully processed.

success_url may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use success_url="/polls/{slug}/" to redirect to a URL composed out of the slug field on a model.

ModelFormMixin

Methods:-

get_form_class() - Retrieve the form class to instantiate. If form_class is provided, that class will be used. Otherwise, a ModelForm will be instantiated using the model associated with the queryset, or with the model, depending on which attribute is provided.

get_form_kwargs() - Add the current instance (self.object) to the standard get_form_kwargs().

get_success_url() - Determine the URL to redirect to when the form is successfully validated. Returns django.views.generic.edit.ModelFormMixin.success_url if it is provided; otherwise, attempts to use the get_absolute_url() of the object.

form_valid(form) - Saves the form instance, sets the current object for the view, and redirects to get_success_url().

ModelFormMixin

Methods:-

form_invalid(form) - Renders a response, providing the invalid form as context.

Creating Model for CreateView

models.py

```
from django.db import models
from django.urls import reverse
class Student(models.Model):
 name = models.CharField(max_length=70)
 roll = models.IntegerField()
 def get_absolute_url(self):
     return reverse("thankyou")
     # return reverse("studentdetail", kwargs={"pk": self.pk})
```

urls.py

```
urlpatterns = [       path('student/', views.StudentCreateView.as_view(), name='studentform’),
  # path('model_detail/<int:pk>/', views.StudentDetailView.as_view(), name='studentdetail'),
]
```

CreateView with Default Template

views.py

```
from django.views.generic.edit import CreateView
from .models import Student
class StudentCreateView(CreateView):
        model = Student
        fields = (‘name’, ‘roll’) # fields = ‘__all__’
```

urls.py

```
urlpatterns = [       path('student/', views.StudentCreateView.as_view(), name='studentform’),
  # path('model_detail/<int:pk>/', views.StudentDetailView.as_view(), name='studentdetail'),
]
```

Default Template should be: student_form.html

CreateView with Custom Template

views.py

```
from django.views.generic.edit import CreateView
from .models import Student
class StudentCreateView(CreateView):
        model = Student
        fields = (‘name’, ‘roll’)
        template_name = 'school/student.html’
```

urls.py

```
urlpatterns = [       path('student/', views.StudentCreateView.as_view(), name='studentform’),
  # path('model_detail/<int:pk>/', views.StudentDetailView.as_view(), name='studentdetail'),
]
```

UpdateView

django.views.generic.edit.UpdateView

A view that displays a form for editing an existing object, redisplaying the form with validation errors (if there are any) and saving changes to the object. This uses a form automatically generated from the object's model class (unless a form class is manually specified).

This view inherits methods and attributes from the following views:

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.edit.BaseUpdateView
- django.views.generic.edit.ModelFormMixin
- django.views.generic.edit.FormMixin
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.edit.ProcessFormView
- django.views.generic.base.View

UpdateView

Attributes:-

template_name_suffix - The UpdateView page displayed to a GET request uses a template_name_suffix of '_form’.

object - When using UpdateView you have access to self.object, which is the object being updated.

UpdateView with Default Template

views.py

```
from django.views.generic.edit import UpdateView
from .models import Student
class StudentUpdateView(UpdateView):
        model = Student
        fields = (‘name’, ‘roll’)
```

urls.py

```
urlpatterns = [
path('updatestudent/<int:pk>', views.StudentUpdateView.as_view(), name='update_student’),
]
```

Default Template should be: student_form.html This is same as create template file.

UpdateView with Custom Template

views.py

```
from django.views.generic import UpdateView
from .models import Student
class StudentUpdateView(UpdateView):
 model = Student
 fields = ['name', 'roll']
 template_name = 'school/student.html’
```

urls.py

```
urlpatterns = [
 path('updatestudent/<int:pk>', views.StudentUpdateView.as_view(), name='update_student’),
]
```

Note The Create View Template File must be same

DeleteView

django.views.generic.edit.DeleteView

A view that displays a confirmation page and deletes an existing object. The given object will only be deleted if the request method is POST. If this view is fetched via GET, it will display a confirmation page that should contain a form that POSTs to the same URL.

This view inherits methods and attributes from the following views:

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.edit.BaseDeleteView
- django.views.generic.edit.DeletionMixin
- django.views.generic.detail.BaseDetailView
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.base.View

DeleteView

Attribute:-

template_name_suffix - The DeleteView page displayed to a GET request uses a template_name_suffix of '_confirm_delete’.

DeletionMixin

django.views.generic.edit.DeletionMixin

Enables handling of the DELETE http action.

Attributes:-

success_url - The url to redirect to when the nominated object has been successfully deleted.

success_url may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use success_url="/parent/{parent_id}/" to redirect to a URL composed out of the parent_id field on a model.

Methods:-

delete(request, *args, **kwargs) - Retrieves the target object and calls its delete() method, then redirects to the success URL.

get_success_url() - Returns the url to redirect to when the nominated object has been successfully deleted. Returns success_url by default.

DeleteView with Default Template

views.py

```
from django.views.generic.edit import DeleteView
From Django.urls import reverse_lazy
from .models import Student
class StudentDeleteView(DeleteView):
        model = Student
        success_url = reverse_lazy(‘add_student’)          # After Delete Redirect to detail page
```

urls.py

```
urlpatterns = [
 path('deletestudent/<int:pk>', views.StudentDeleteView.as_view(), name='delete_student’),
]
```

Default Template should be: student_confirm_delete.html

DeleteView with Default Template

Student_confirm_delete.py

```
<body>
 <h1>Are you Sure ?</h1>
 <form action="" method="post">
  {% csrf_token %}
  <input type="submit" value="Delete">
  <a href="{% url 'add_student' %}">Cancel</a>
 </form>
</body>
```

DeleteView with Custom Template

views.py

```
from django.views.generic.edit import DeleteView
from django.urls import reverse_lazy
from .models import Student
class StudentDeleteView(DeleteView):
        model = Student
        success_url = reverse_lazy(‘add_student’)          # After Delete Redirect to detail page
        template_name = ‘school/studentdeleteconfirmation.html’
```

urls.py

```
urlpatterns = [
 path('deletestudent/<int:pk>', views.StudentDeleteView.as_view(), name='delete_student’),
]
```

CBV and FBV

CBV

It is not powerful.

Its Wrapper for FBV to hide complexity

This is used in most generic view

Flow of execution is unexpected the flow is defined internally

For Get Request define get() and for POST define post ()

Its easy to use and have reusability.

Model Related Class Based View

ListView – To List all Record. This will look for default template file named ModelClassName_list.html and default context object named ModelClassName_list. We can also specific our own template and context file using template_name = student.html as well context by context_object_name = list_of_student

DetailView – To get details of a particular Record. This will look for default template file named ModelClassName_detail.html and default context object named ModelClassName or object. We can also specific our own template and context file using template_name = student.html as well context by context_object_name = list_of_student

CreateView – To Create Record

This will look for default template file named ModelClassName_form.html. We can also specific our own template and context file using template_name = studentform.html. We also need to define get_absolute_url so it will redirected after insertion.

Model Related Class Based View

UpdateView – To Update Record. This is very similar to create infact it will use CreateView default template file. This will look for default template file named ModelClassName_detail.html . We can also specific our own template and context file using template_name = studentform.html. We also need to define get_absolute_url so it will redirected after insertion.

DeleteView – To Delete Record

This will look for default template file named ModelClassName_confirm_delete.html and default context object named ModelClassName or object. We can also specific our own template file using template_name = studentdeleteconfirmation.html