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

CreateView should be used when you need a form on the page and need to do a db insertion on submission of a valid form.

[#](https://www.agiliq.com/blog/2019/01/django-createview/#createview-is-better-than-vanilla-view)CreateView is better than vanilla View

We will first write a vanilla view by subclassing **View**, and then modify the view to subclass **CreateView** instead of **View**.

CreateView is better than vanilla View in following ways:

* Avoid boilerplate code
* Succinct and more maintainable code.

[#](https://www.agiliq.com/blog/2019/01/django-createview/#vanilla-view)Vanilla View

We want to create a page with a book creation form.

# books/models.py

class Book(models.Model):

title = models.CharField(max\_length=100)

isbn = models.CharField(max\_length=100, unique=True)

is\_published = models.BooleanField(default=True)

def \_\_str\_\_(self):

return self.title

# books/forms.py

class BookCreateForm(forms.ModelForm):

class Meta:

model = Book

Vanilla view looks like:

# books/views.py

class BookCreateView(CreateView):

def get(self, request, \*args, \*\*kwargs):

context = {'form': BookCreateForm()}

return render(request, 'books/book-create.html', context)

def post(self, request, \*args, \*\*kwargs):

form = BookCreateForm(request.POST)

if form.is\_valid():

book = form.save()

book.save()

return HttpResponseRedirect(reverse\_lazy('books:detail', args=[book.id]))

return render(request, 'books/book-create.html', {'form': form})

Template code looks like:

<!--books/templates/books/book-create.html-->

<form action="." method="POST">

{% csrf\_token %}

<table>

</table>

<button type="submit">SUBMIT</button>

</form>

With proper urlpattern, you should be able to see the book creation form.

from django.urls import path

from . import views

app\_name = 'books'

urlpatterns = [

path('create/', views.BookCreateView.as\_view(), name='create'),

path('<int:pk>/', views.BookDetailView.as\_view(), name='detail'),

]

[#](https://www.agiliq.com/blog/2019/01/django-createview/#using-createview)Using CreateView

Vanilla view has a lot of boilerplate code.

Any object creation view will have a get() implementation for creating context and rendering the response. Similarly object creation view will have a post() implementation to do .save(). CreateView, which is a generic class based view, can avoid this boilerplate code.

class BookCreateView(CreateView):

template\_name = 'books/book-create.html'

form\_class = BookCreateForm

This change also needs that a get\_absolute\_url() be defined on the object which is being created. So we need to provide a get\_absolute\_url() on model Book.

class Book(models.Model):

# More code

def get\_absolute\_url(self):

return reverse('books:detail', args=[self.id])

Refresh the page and you should still be able to achieve everything that was possible with vanilla view.

As you would have noticed, using a CreateView helped us avoid boilerplate get() and post() implementation. The code looks much more succinct as it only has few class attributes and there isn’t any function implementation.

[#](https://www.agiliq.com/blog/2019/01/django-createview/#adding-initial-data-to-createview)Adding initial data to CreateView

Assume we want to populate form’s title field with some initial data.

Modify BookCreateView to look like:

class BookCreateView(CreateView):

template\_name = 'books/book-create.html'

form\_class = BookCreateForm

def get\_initial(self, \*args, \*\*kwargs):

initial = super(BookCreateView, self).get\_initial(\*\*kwargs)

initial['title'] = 'My Title'

return initial

This code has better separation of concern. There is a separate method for dealing with initial data.

Had we used a vanilla view, initial data code would have been part of get().

[#](https://www.agiliq.com/blog/2019/01/django-createview/#adding-form-kwargs-to-createview)Adding form kwargs to CreateView

Let’s add a user field to Book to track the user who creates a Book.

class Book(models.Model):

title = models.CharField(max\_length=100)

isbn = models.CharField(max\_length=100, unique=True)

is\_published = models.BooleanField(default=True)

user = models.ForeignKey(User, on\_delete=models.CASCADE, \*\*NULL\_AND\_BLANK)

def \_\_str\_\_(self):

return self.title

def get\_absolute\_url(self):

return reverse('books:detail', args=[self.id])

Assume you don’t want to allow a user to create two books with same title. The title should be unique per user.

This validation needs writing a clean\_title() method which would look like:

class BookCreateForm(forms.ModelForm):

class Meta:

model = Book

exclude = ('user',)

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.user = kwargs.pop('user')

super(BookCreateForm, self).\_\_init\_\_(\*args, \*\*kwargs)

def clean\_title(self):

title = self.cleaned\_data['title']

if Book.objects.filter(user=self.user, title=title).exists():

raise forms.ValidationError("You have already written a book with same title.")

return title

This needs that a user be supplied from view during form creation. This is where CreateView.get\_form\_kwargs() come into picture. Modify the view to look like:

class BookCreateView(CreateView):

template\_name = 'books/book-create.html'

form\_class = BookCreateForm

def form\_valid(self, form):

self.object = form.save(commit=False)

self.object.user = self.request.user

self.object.save()

return HttpResponseRedirect(self.get\_success\_url())

def get\_initial(self, \*args, \*\*kwargs):

initial = super(BookCreateView, self).get\_initial(\*\*kwargs)

initial['title'] = 'My Title'

return initial

def get\_form\_kwargs(self, \*args, \*\*kwargs):

kwargs = super(BookCreateView, self).get\_form\_kwargs(\*args, \*\*kwargs)

kwargs['user'] = self.request.user

return kwargs

After this any logged in user wouldn’t be able to create two Books with same title.