Understanding Django Views

(*The Heart of Django Development*)

Satyendra Gautam

When it comes to Django, ***views*** **are the heart of development**—this is where your application's core logic and business rules live.

Introduction to Django Views

When working with Django, views are where your application's core logic is implemented. There are two primary ways to handle requests in Django: Function-Based Views (FBVs) and Class-Based Views (CBVs). Let's break down what each of them means and how they differ.

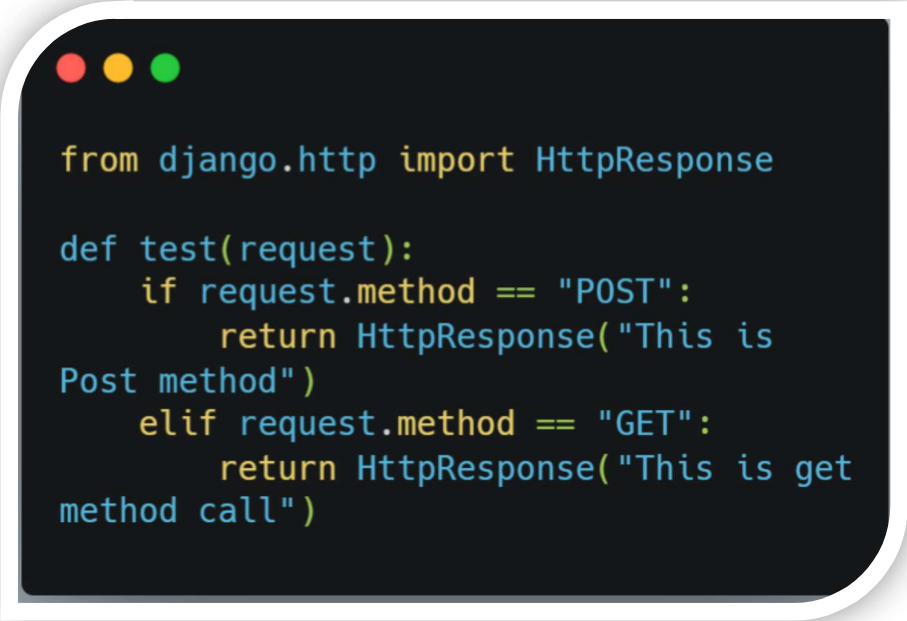
Types Of Django Views

1. Function Based View
2. Class Based View
    - Base View
    - Generic View

Function-Based Views (FBVs)

A Function-Based View is simply a Python function that takes an *HTTP request* as input and returns a response. This response can be a rendered HTML template, a *HttpResponse*, a *JSON response*, or even an error page.

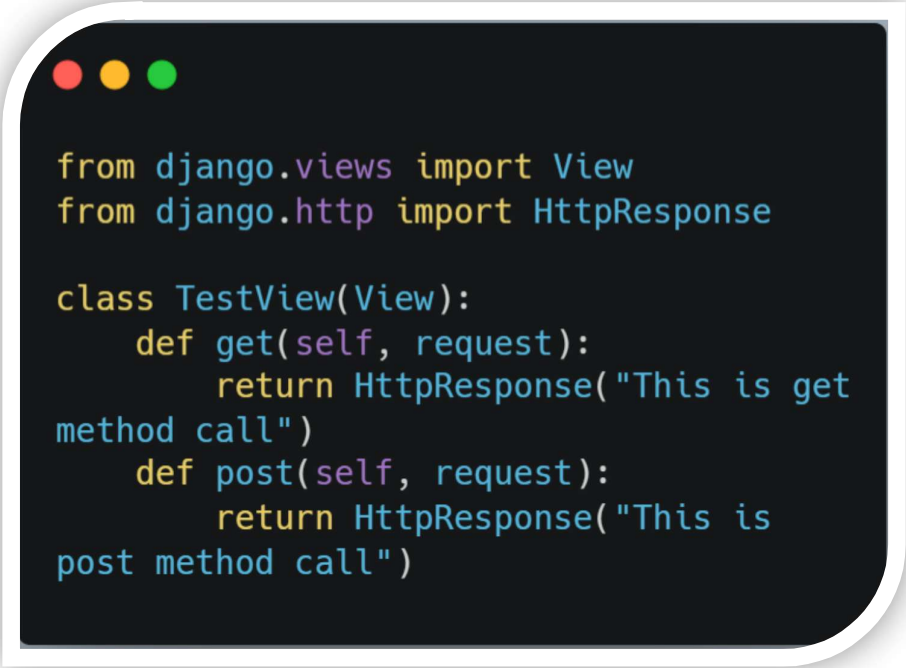
FBVs typically use conditional statements to handle different HTTP methods
(like GET, POST, PUT, PATCH, DELETE).



```
from django.http import HttpResponse

def test(request):
    if request.method == "POST":
        return HttpResponse("This is
Post method")
    elif request.method == "GET":
        return HttpResponse("This is get
method call")
```

Class-Based Views (CBVs)

A Class-Based View is a *Python class* that inherits from Django's *built-in View* class or one of the *generic views*. Instead of using conditional statements inside a single function, CBVs provide separate methods like get(), post(), put(), patch(), and delete() to handle each HTTP method individually.



```python
from django.views import View
from django.http import HttpResponse

class TestView(View):
    def get(self, request):
        return HttpResponse("This is get
method call")
    def post(self, request):
        return HttpResponse("This is
post method call")
```

Types of Class-Based Views in Django

Django provides two main types of *Class-Based Views (CBVs):* ***Base Views*** *and* ***Generic Views****.

Base Views

Base Views are foundational classes *View* and *TemplateView.*

These views give you complete control over handling requests and crafting responses. They're ideal when you want to fully understand or customize the request-response flow.

Using Base Views, you manually define methods like get(), post(), etc., giving you a clear picture of how your view behaves for different HTTP methods.

Generic View

Generic Views are powerful, pre-built views provided by Django to simplify common tasks like displaying lists, creating new objects, updating, or deleting them.

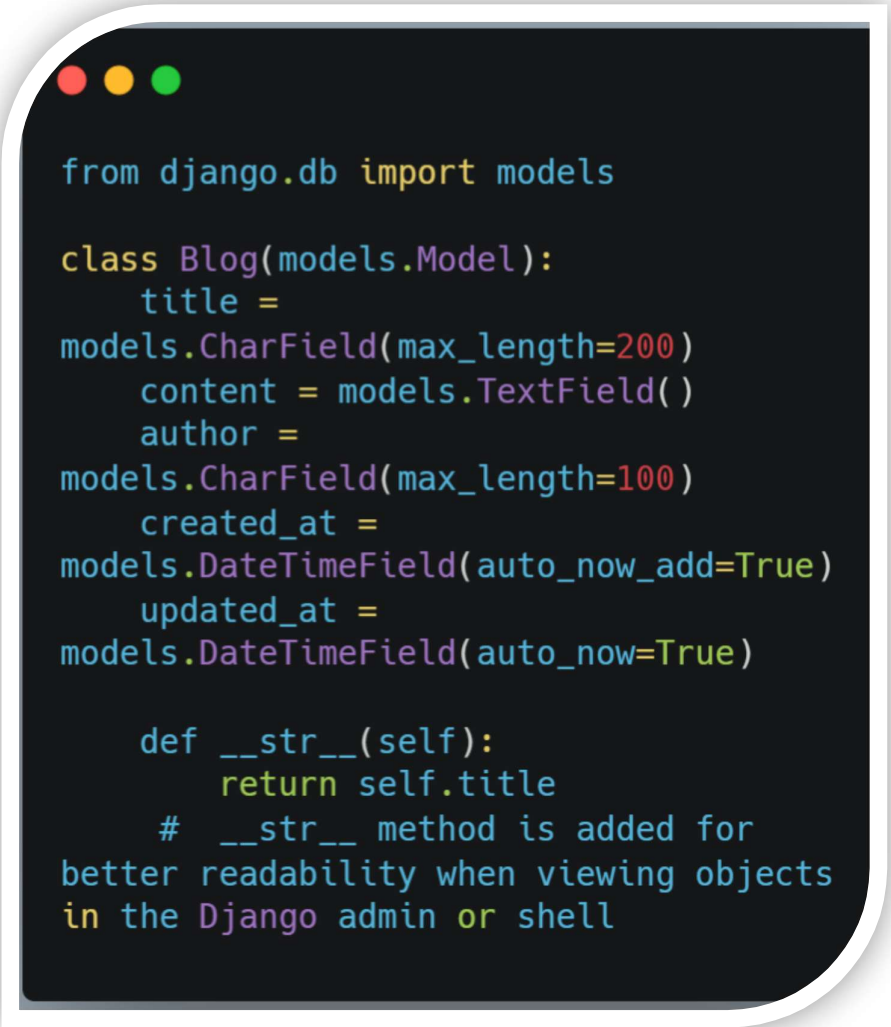
ListView, CreateView, UpdateView, DetailView, DeleteView .

Note: FBVs vs. CBVs

It’s important to understand that **Class-Based Views do not replace Function-Based Views**, and vice versa.

Setting Up the Blog Model (for CRUD Operations)

By the help of this model we perform crud operations

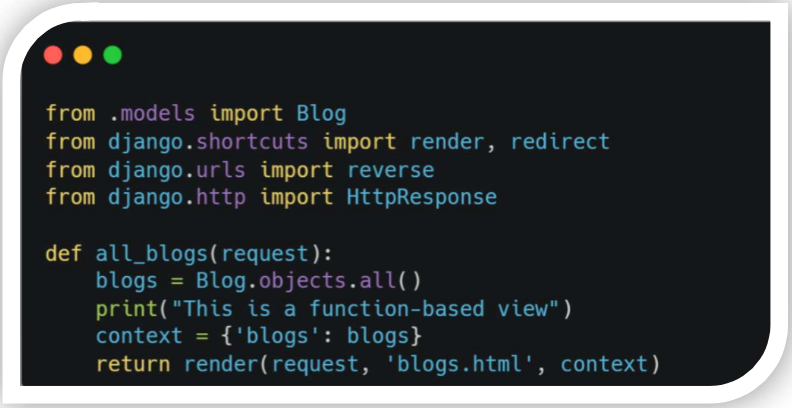


```
from django.db import models

class Blog(models.Model):
    title =
models.CharField(max_length=200)
    content = models.TextField()
    author =
models.CharField(max_length=100)
    created_at =
models.DateTimeField(auto_now_add=True)
    updated_at =
models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.title
     #  __str__ method is added for
better readability when viewing objects
in the Django admin or shell
```

CRUD with *Function-Based Views* (FBVs)

Let's start with the **READ** operation using Function-Based Views. This example demonstrates how to retrieve all blog entries from the database and pass them to a template for rendering.

1. Read (GET) – Display All Blogs



```python
from .models import Blog
from django.shortcuts import render, redirect
from django.urls import reverse
from django.http import HttpResponse

def all_blogs(request):
    blogs = Blog.objects.all()
    print("This is a function-based view")
    context = {'blogs': blogs}
    return render(request, 'blogs.html', context)
```

you can also used json response to retrieve data in browser

```python
def all_blogs_json(request):
    blogs = Blog.objects.all()
    data = [
        {
            'id': blog.id,
            'title': blog.title,
            'content': blog.content,
            'author': blog.author,
            'created_at': blog.created_at,
            'updated_at': blog.updated_at,
        }
        for blog in blogs
    ]
    # safe=False for Non-Dictionary Data.
    return JsonResponse(data, safe=False)
```

Why Do We Need safe=False in JsonResponse? 🤔

By default, *JsonResponse* expects the data passed to it to be a dictionary. If we want to return a list or another data structure like Python Lists, we need to set the safe parameter to False.

safe=False *tells Django:*
"Yes, I know I'm returning a list instead of a dictionary—and I'm doing it intentionally. It's safe to send this response."

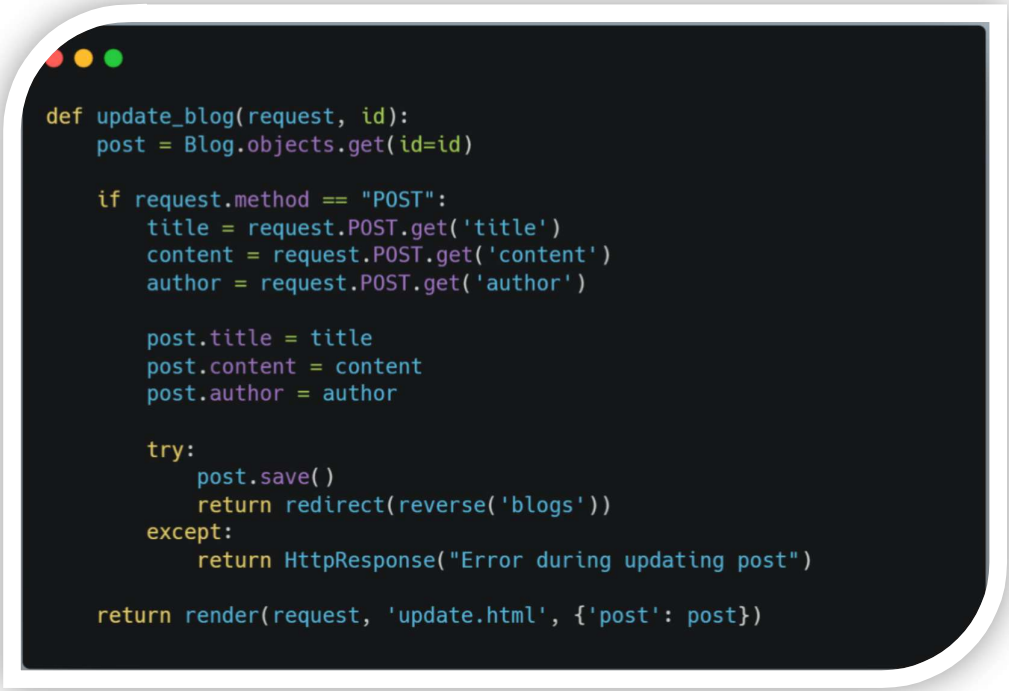
2. **Create a Blog**

To add a new blog post, we use a function-based view that handles both GET (to show the form) and POST (to save the form data).

```
def create_blog(request):
    if request.method == "POST":
        # Extracting form data from POST request
        title = request.POST.get('title')
        content = request.POST.get('content')
        author = request.POST.get('author')

        try:
            # Creating a new Blog entry in the database
            Blog.objects.create(title=title, content=content,
author=author)

            # Redirect to blog list page using reverse
            return redirect(reverse('blogs'))
            # this is works same as
            # return redirect('/')

        except Exception as e:
            # Handling unexpected errors
            return HttpResponse(f"Error while creating blog: {e}")
    # For GET request, render the form template
    return render(request, 'create.html')
```

3. Update (UPDATE) – Modify an Existing Blog Post

To update a blog, the first step is to fetch the blog post using its unique identifier — either the id or the primary key (pk). Once we retrieve the blog, we populate an update form with the existing data using Django's context.



```python
def update_blog(request, id):
    post = Blog.objects.get(id=id)

    if request.method == "POST":
        title = request.POST.get('title')
        content = request.POST.get('content')
        author = request.POST.get('author')

        post.title = title
        post.content = content
        post.author = author

        try:
            post.save()
            return redirect(reverse('blogs'))
        except:
            return HttpResponse("Error during updating post")

    return render(request, 'update.html', {'post': post})
```

🤔 Bro, Why Do We Use POST Instead of PUT, PATCH, or UPDATE in Django Forms?

The simple answer is:

By default, HTML forms only support *GET and POST methods.*

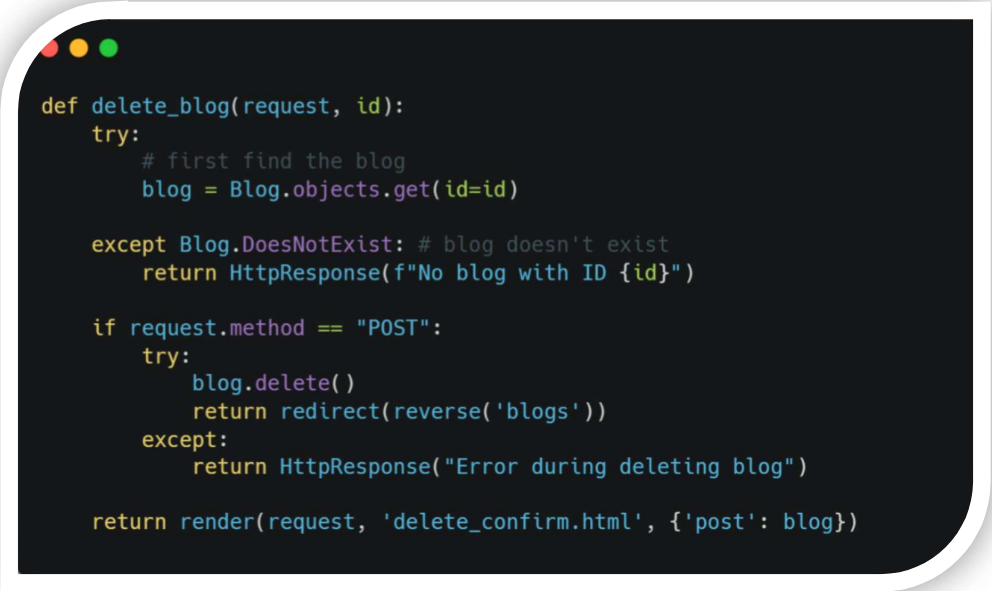
That's why, when working with Django's traditional (non-API) views and templates, we use POST for operations like create or update.

Delete (DELETE) – Remove an Existing Blog Post

You can delete a blog post in two ways:

1. **Directly via URL** – Just visit:
   http://127.0.0.1:8000/delete/7

   Where 7 is the ID of the blog you want to delete.)
2. **With a Confirmation Page** – This is a safer method, especially in real-world apps, to avoid accidental deletions.



```
def delete_blog(request, id):
    try:
        # first find the blog
        blog = Blog.objects.get(id=id)

    except Blog.DoesNotExist: # blog doesn't exist
        return HttpResponse(f"No blog with ID {id}")

    if request.method == "POST":
        try:
            blog.delete()
            return redirect(reverse('blogs'))
        except:
            return HttpResponse("Error during deleting blog")

    return render(request, 'delete_confirm.html', {'post': blog})
```

✅ Now that we've successfully built all CRUD operations using Function-Based Views (FBVs), it's time to level up!

we'll explore how to implement the same CRUD functionality using Class-Based Views (CBVs) and Generic Views — Django's powerful tools that make your code more structured, reusable, and concise.

CRUD with Class-Based Views (CBVs)

In function-based views, we typically check if the request is GET or POST, and based on the request type, we process the appropriate response. By default, the GET method is used for rendering views.

In class-based views, we handle this by defining *def get()* for GET requests and *def post()* for POST operations.

1. Read (GET)

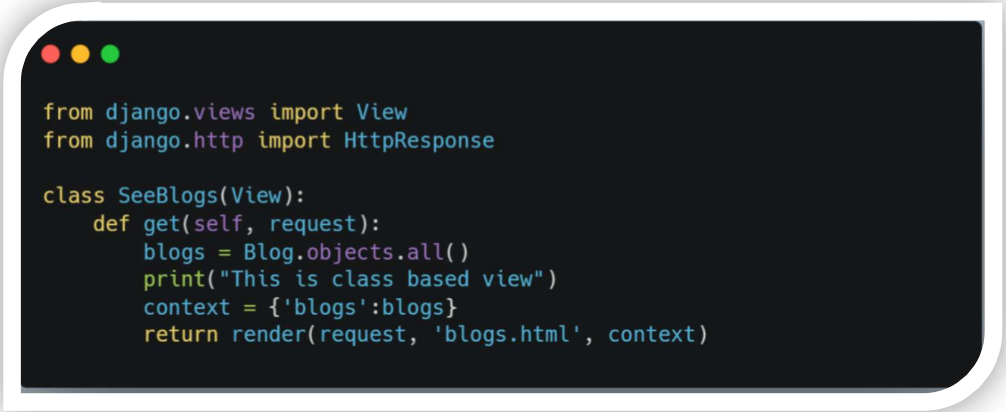
Why do we use *.as_view()* **in Class-Based Views?**

we are defining methods (get, post, etc.) **inside a class**, not a function. But Django's urlpatterns expects a **callable view function**, not a class.

🤔 **So what does** .as_view() **do?**

✅ .as_view() is a **built-in method** in Django that:

- Converts our class (SeeBlogs) into a callable view function (like FBVs).
- Internally it maps the incoming HTTP method (GET, POST, etc.) to the respective class method (get(), post(), etc.).

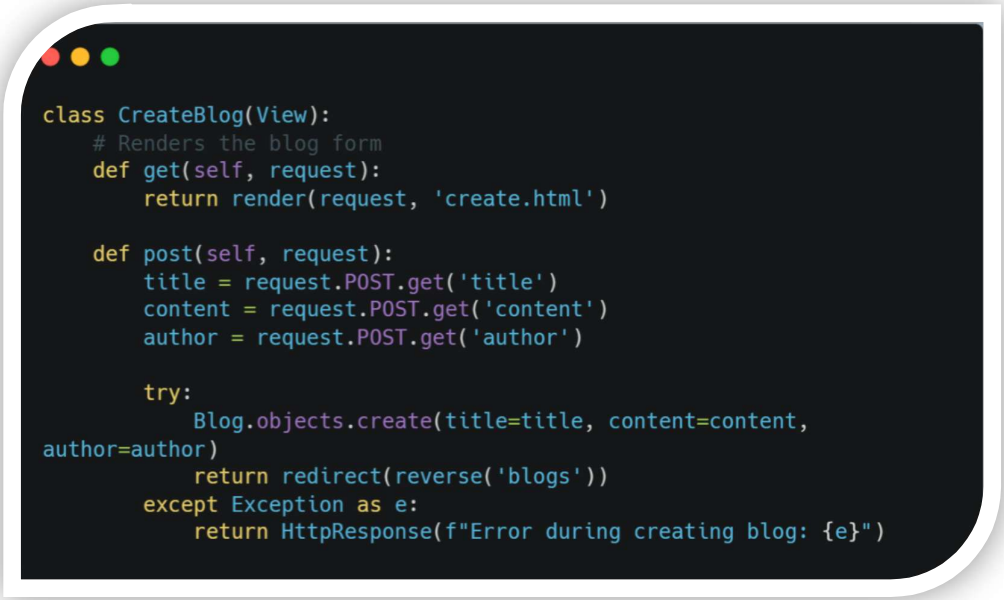


```python
from django.views import View
from django.http import HttpResponse

class SeeBlogs(View):
    def get(self, request):
        blogs = Blog.objects.all()
        print("This is class based view")
        context = {'blogs':blogs}
        return render(request, 'blogs.html', context)
```

Create (POST) Using Class-Based Views (CBVs)

Creating a new blog using CBVs is a bit more structured compared to FBVs.

In function-based views, the browser by default uses the GET method to render the form — so we usually check the method inside a single function.
But in CBVs, things are cleaner and more modular: we explicitly define separate methods for GET (to render the form) and POST (to handle form submission).



```
class CreateBlog(View):
    # Renders the blog form
    def get(self, request):
        return render(request, 'create.html')

    def post(self, request):
        title = request.POST.get('title')
        content = request.POST.get('content')
        author = request.POST.get('author')

        try:
            Blog.objects.create(title=title, content=content,
author=author)
            return redirect(reverse('blogs'))
        except Exception as e:
            return HttpResponse(f"Error during creating blog: {e}")
```

Update Blog using CBV (Class-Based View)

Just like we did in the create view, here we need to:

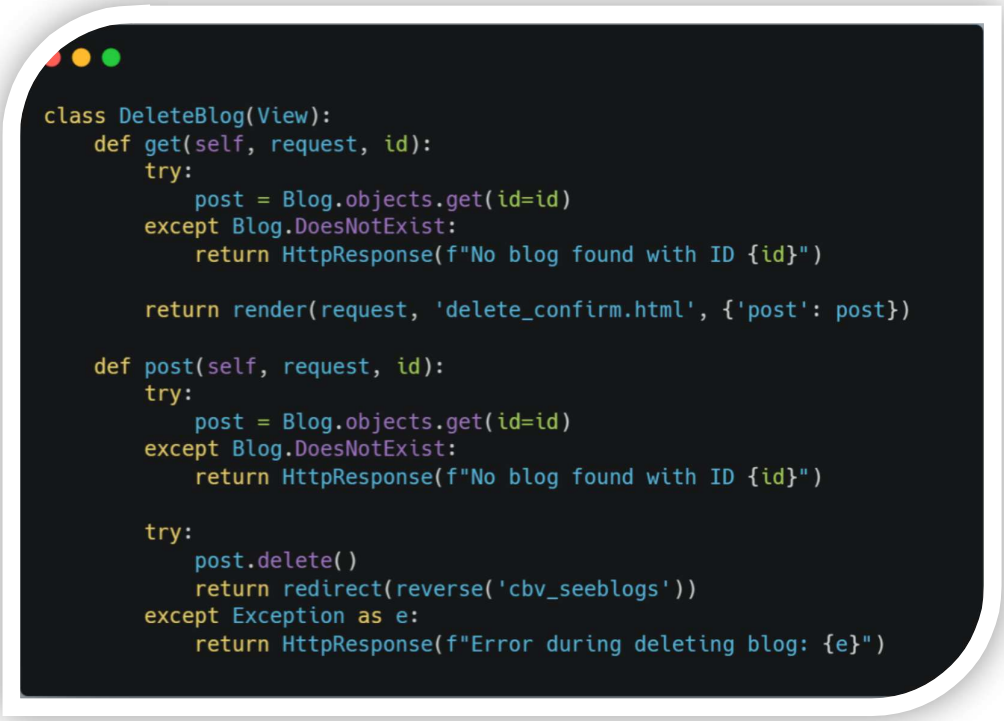
1. Render the update form pre-filled with the blog's existing data.
2. Handle the form submission using a POST request to update the blog.

```python
class UpdateBlog(View):
    # render the update form
    def get(self, request, id):
        try:
            post = Blog.objects.get(id=id)
        except Blog.DoesNotExist:
            return HttpResponse(f"No blog post found with id: {id}")

        return render(request, 'update.html', {'post': post})

    def post(self, request, id):
        try:
            post = Blog.objects.get(id=id)
        except Blog.DoesNotExist:
            return HttpResponse(f"No blog post found with id: {id}")

        post.title = request.POST.get('title')
        post.content = request.POST.get('content')
        post.author = request.POST.get('author')

        try:
            post.save()
            return redirect(reverse('blogs'))
        except Exception as e:
            return HttpResponse(f"Error during updating: {e}")
```

Delete Blog using CBV (Class-Based View)

Deleting a blog using CBV is straightforward. You just need to:

1. Get the blog post using its id.
2. Render a confirmation page using a GET request.
3. If confirmed (POST request), delete the blog post.



```python
class DeleteBlog(View):
    def get(self, request, id):
        try:
            post = Blog.objects.get(id=id)
        except Blog.DoesNotExist:
            return HttpResponse(f"No blog found with ID {id}")

        return render(request, 'delete_confirm.html', {'post': post})

    def post(self, request, id):
        try:
            post = Blog.objects.get(id=id)
        except Blog.DoesNotExist:
            return HttpResponse(f"No blog found with ID {id}")

        try:
            post.delete()
            return redirect(reverse('cbv_seeblogs'))
        except Exception as e:
            return HttpResponse(f"Error during deleting blog: {e}")
```

CRUD with Generic Views(GBV)

Read (GET)

Reading all blog posts with Generic Class-Based Views (GCBVs) in Django is super elegant and requires minimal code. Django provides a pre-built class called ListView that does the heavy lifting for us.

✅ What does ListView do?

Under the hood, ListView:

- Automatically fetches all records using *Model.objects.all().*
- Prepares the data for rendering in a template.
- Makes your life easier by eliminating the need to manually write a get() method.

```python
from django.views.generic import ListView
from .models import Blog

class SeeblogsGeneric(ListView):
    model = Blog
    template_name = 'blogs.html'
    context_object_name = 'blogs'

    # no need to define def get(): .........
```

🔍 Behind the Scenes

If you open Django's source code for *ListView*, you'll find that it internally uses something like this *context['blogs'] = Blog.objects.all()*

Create (POST) Using Generic View

Creating a new blog post becomes **super simple** with Django's generic views — and that's the power of Generic View ✨

Instead of manually handling get and post like we do in CBVs, Django's *CreateView* handles it all under the hood. You just need to tell it **what to work with**, and it does the rest.

```python
from django.views.generic import CreateView

class CreateBlogGeneric(CreateView):
    model = Blog
    # Reusing the same template
    template_name = 'create.html'
    fields = ['title', 'content', 'author']
     # Redirect after successful post
    success_url = reverse_lazy('blogs')
```

reverse vs reverse_lazy – Why It Matters in CBV vs Generic View

You might be wondering:
*"**Bro , Why are we using** reverse() in FBVs and CBVs, but reverse_lazy() in Generic Views?"*

🧭 Both reverse() and reverse_lazy() are used to **redirect the user after a successful operation** — like after creating, deleting or updating a blog.

🕓 reverse() – Do it Now

- Used in **function-based views (FBVs)** and **regular class-based views (CBVs)** inside methods like post().
- It **immediately resolves** the URL when the function runs.
- Works perfectly fine here because the view function/class method is already being executed.

reverse_lazy() – Do it Later (Reverse Karo Bad me Jab Jarurat Ho)

- Used in **generic class-based views (GCBVs)** like CreateView, UpdateView, DeleteView etc.
- These views are **set up at class definition time**, before any method like post() is called.
- If you use reverse() here, Django tries to resolve the URL **too early**, which can lead to errors or unexpected behavior.
- So we use reverse_lazy() — think of it like saying:
  🧠 *"I'll reverse this URL later… only when it's actually needed."*

Update Blog using GBV (Generic View) – UpdateView

Updating a blog post with **Generic Views** is super convenient. Just like CreateView, you only need to specify a few key attributes:

- The model you're working with
- The template_name for the update form
- The fields you want to allow editing
- The success_url to redirect after successful update

```python
from django.views.generic import UpdateView

class UpdateBlogGeneric(UpdateView):
    model = Blog
    template_name = 'update.html'
    fields = ['title', 'content', 'author']
    context_object_name = "post"
    success_url = reverse_lazy('gen_seeblogs')
```

Delete Blog using GCBV (Generic Class-Based View)

Deleting a blog using Django's built-in DeleteView is extremely efficient. Unlike CBVs where we manually write
both GET and POST methods, DeleteView handles everything behind the scenes.

```python
from django.views.generic import DeleteView

class DeleteBlogGeneric(DeleteView):
    model = Blog
    template_name = 'delete_confirm.html'
    context_object_name = 'post'
    success_url = reverse_lazy('gen_seeblogs')
```

When to Use Class-Based Views vs Generic Views?

One of the most common questions while learning Django is: **"Should I use a Class-Based View (CBV) or a Generic View?"**

Use Class-Based Views (CBVs) when:

- You need **full control** over the request/response cycle.
- You want to **understand Django internals** and how views process different HTTP methods like GET, POST, etc.
- You're in **learning mode** and want to master how Django handles requests.

Use Generic Class-Based Views (GCBVs) when:

- You want to **write less code** and let Django handle the common logic.
- Your view logic follows standard CRUD patterns (like listing, creating, updating, deleting).
- You want **clean, DRY, and maintainable** code.

As a Smart Django Developer, Should You Use FBV or CBV?

This is a common question every Django developer faces. There's no one-size-fits-all answer, but here's how you can make a smart decision

When to Use Function-Based Views (FBVs)

FBVs are simple Python functions that take a request and return a response.

- **Simplicity and Clarity**
- **Explicit and Easy to Read**
- **Quick Prototyping**
- **Team Preference**

🧠 When to Use Class-Based Views (CBVs)

CBVs are Python classes that organize views into methods (get(), post(), etc.).

- **Reusability**
- **Better Organization**
- **Django's Generic Views**

Final Thoughts

In this Series, we've walked through every approach to implementing CRUD in Django — from the traditional **Function-Based Views (FBVs)**, to the structured **Class-Based Views (CBVs)**, and finally the powerful and concise **Generic Views**.

Each method has its own strengths:

- **FBVs** offer simplicity and complete control.
- **CBVs** bring better organization and reusability.
- **Generic Views** cut down on boilerplate, helping you move fast with less code.

🙏 Final Note

If you've made it this far — **congratulations**!
You now own this PDF, and you're **completely free to use, study, share, or remix** it however you like.

- If this guide helped you understand Djnago View better, I'm truly glad.
- I don't ask for likes, shares, or reposts —
  but if you found it helpful, I'd really appreciate a kind comment or suggestion.

If you spot any mistakes — **please forgive me**.
I'm a learner too, and like everyone else, I make mistakes sometimes 😊

Thanks for reading — and keep building, keep learning 💡

— *Satyendra Gautam*