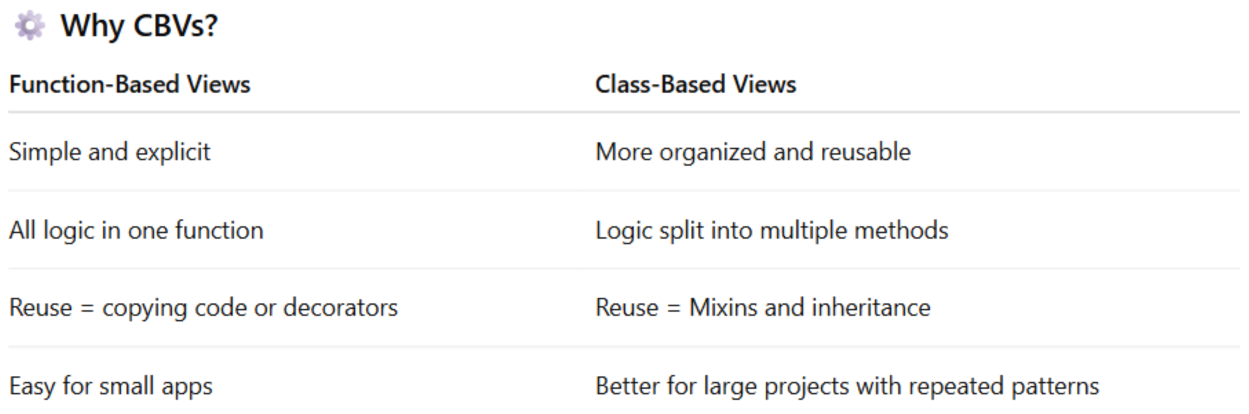
Class-Based Views (CBVs)

🧠 What Are CBVs?

In Django, **views** are the components that receive an HTTP request and return an HTTP response.

You can define views in **two ways:**

- **Function-Based Views (FBVs):**
  - You write logic for GET, POST, etc., inside an `if request.method == "POST":` block.
- **Class-Based Views (CBVs):**
  - You structure logic using **classes and methods** (e.g., `get()`, `post()`, `put()`).



⚙️ Why CBVs?

Function-Based Views	Class-Based Views
Simple and explicit	More organized and reusable
All logic in one function	Logic split into multiple methods
Reuse = copying code or decorators	Reuse = Mixins and inheritance
Easy for small apps	Better for large projects with repeated patterns

Advantages of CBVs:

- Reusable
- Extensible via **Mixins**
- Readable (each HTTP method = one function)
- Encourages DRY (Don't Repeat Yourself) code
- Built-in generic views for CRUD operations

Working:

🧱 3. Anatomy of a CBV

Each CBV is a **class that subclasses Django's View class** (`django.views.View`).

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

class MyView(View):
    def get(self, request):
        return HttpResponse("GET response")

    def post(self, request):
        return HttpResponse("POST response")
```

Django provides “**generic**” **CBVs** for the most common use-cases (CRUD).

View	Purpose
ListView	Display list of objects
DetailView	Display single object details
CreateView	Form to create new object
UpdateView	Form to edit existing object
DeleteView	Confirm and delete object
TemplateView	Render a static HTML page

ListView Example:

🧩 2. Basic Example

python Copy code

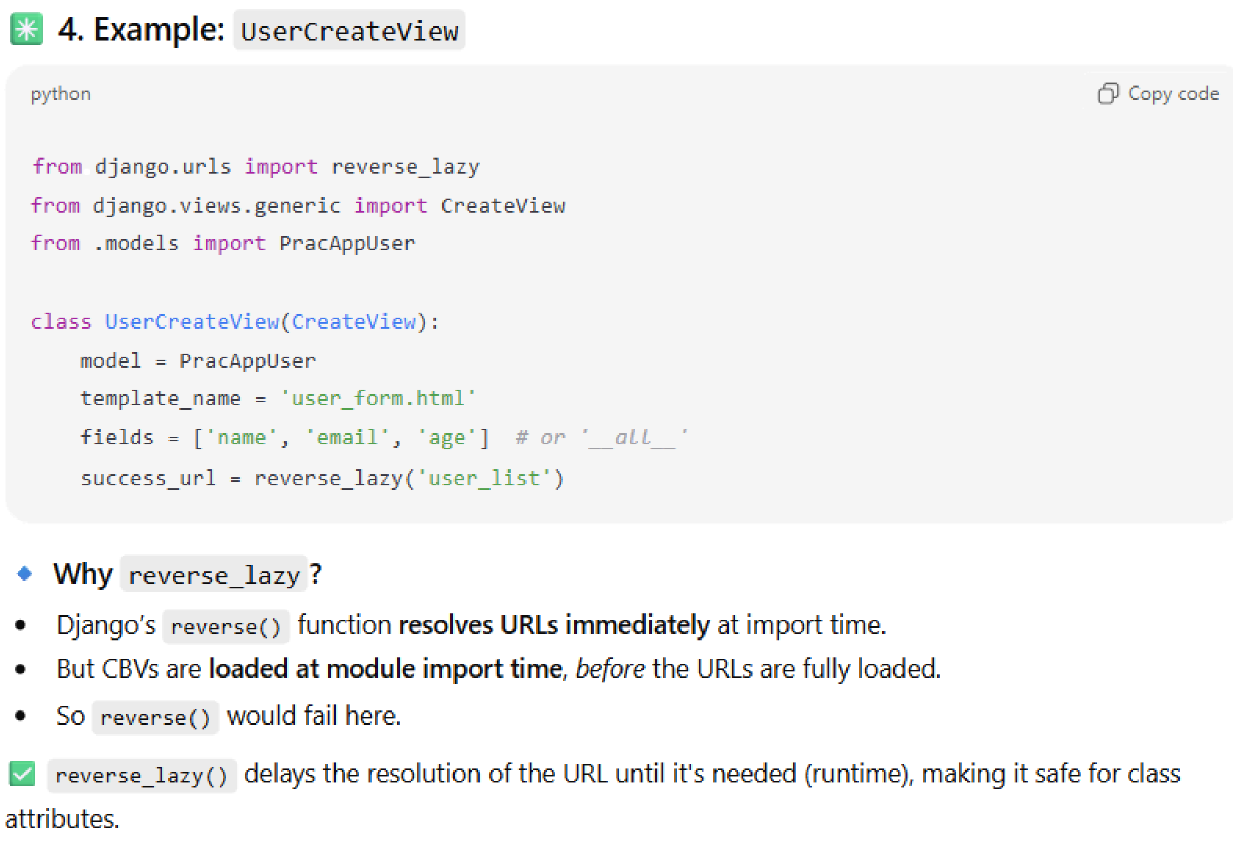
```python
from django.views.generic import ListView
from .models import PracAppUser

class UserListView(ListView):
    model = PracAppUser
    template_name = 'user_list.html'
    context_object_name = 'users'
```

🔹 Explanation:

Property	Purpose
`model`	The model to query from.
`template_name`	The HTML file to render.
`context_object_name`	The variable name for data in the template.
`queryset` *(optional)*	Custom query logic if you want to filter/annotate etc.

Using Queryset: Add this as a method to the UserListView

```python
def get_queryset(self):
    return PracAppUser.objects.filter(is_active=True)
```



✳️ 4. Example: `UserCreateView`

```python
from django.urls import reverse_lazy
from django.views.generic import CreateView
from .models import PracAppUser

class UserCreateView(CreateView):
    model = PracAppUser
    template_name = 'user_form.html'
    fields = ['name', 'email', 'age']  # or '__all__'
    success_url = reverse_lazy('user_list')
```

🔹 **Why `reverse_lazy`?**

- Django's `reverse()` function **resolves URLs immediately** at import time.
- But CBVs are **loaded at module import time**, *before* the URLs are fully loaded.
- So `reverse()` would fail here.

✅ `reverse_lazy()` delays the resolution of the URL until it's needed (runtime), making it safe for class attributes.

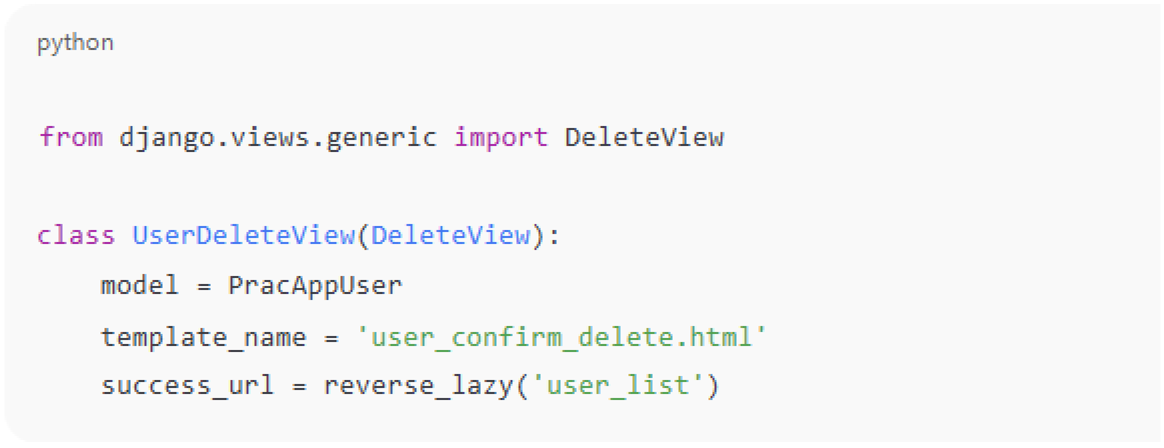
1. Similarly UpdateView as well. Just Inherit UpdateView

#Frontend to show the form for both create and update

```
<body>
    <h1>
        {% if form.instance.pk %}
            Edit User
        {% else %}
            Add New User
        {% endif %}
    </h1>

    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Save</button>
    </form>

    <br>
    <a href="{% url 'user_list' %}">Back to List</a>
</body>
```

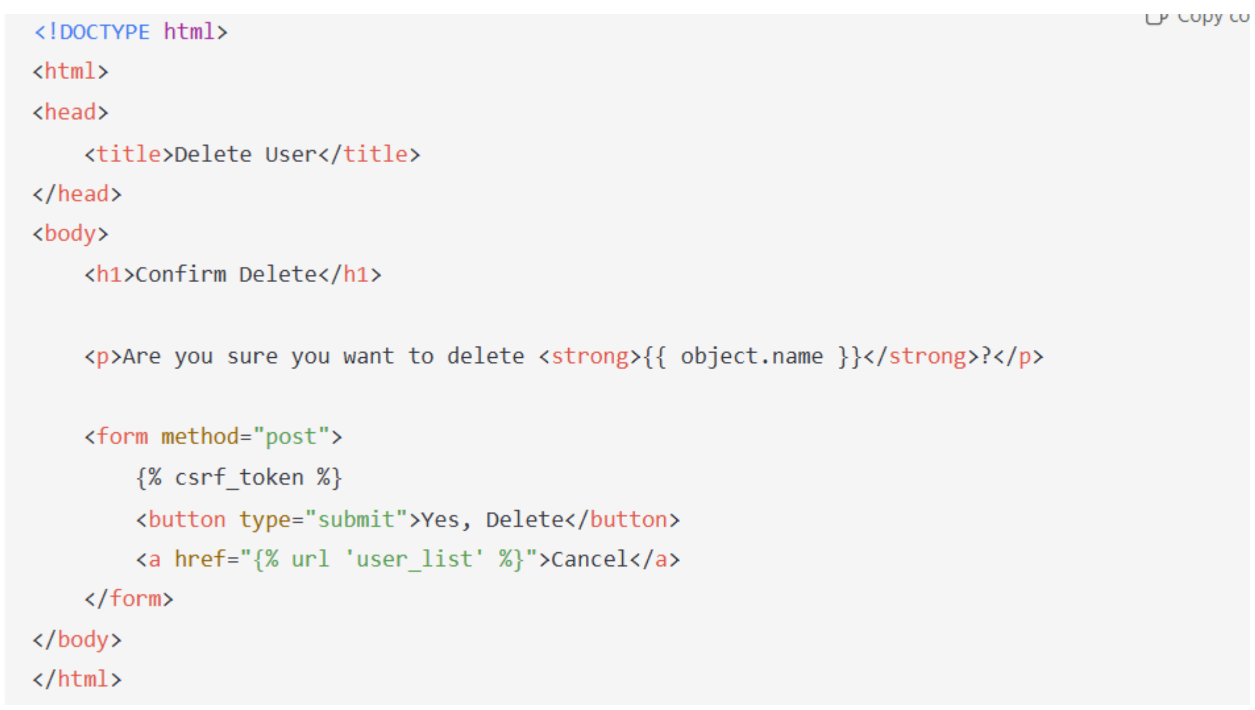
🗑️ 8. DeleteView

Used to **confirm and delete objects.**


python

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

class UserDeleteView(DeleteView):
    model = PracAppUser
    template_name = 'user_confirm_delete.html'
    success_url = reverse_lazy('user_list')
```

Frontend for DeleteView:



```html
<!DOCTYPE html>
<html>
<head>
    <title>Delete User</title>
</head>
<body>
    <h1>Confirm Delete</h1>

    <p>Are you sure you want to delete <strong>{{ object.name }}</strong>?</p>

    <form method="post">
        {% csrf_token %}
        <button type="submit">Yes, Delete</button>
        <a href="{% url 'user_list' %}">Cancel</a>
    </form>
</body>
</html>
```

TemplateView:

Used when you just want to render a static page.

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

class AboutView(TemplateView):
    template_name = 'about.html'
```

🧭 5. Example URL Configuration (`urls.py`)

Make sure your templates match these URL names.

```python
from django.urls import path
from .views import (
    UserListView,
    UserDetailView,
    UserCreateView,
    UserUpdateView,
    UserDeleteView
)

urlpatterns = [
    path('', UserListView.as_view(), name='user_list'),
    path('user/<int:pk>/', UserDetailView.as_view(), name='user_detail'),
    path('user/create/', UserCreateView.as_view(), name='user_create'),
    path('user/<int:pk>/update/', UserUpdateView.as_view(), name='user_update'),
    path('user/<int:pk>/delete/', UserDeleteView.as_view(), name='user_delete'),
]
```

🧩 15. Converting Function-Based Views → Class-Based Views

Function-Based	Class-Based
Use `def`	Use `class`
Logic inside if-blocks	Logic inside methods (`get` , `post`)
Use decorators	Use mixins
Return `render()`	Return `render_to_response()` internally

#Using render_to_response:

```python
from django.views.generic import TemplateView
from .models import Book

class BookListCustomView(TemplateView):
    template_name = 'book_list.html'

    def get(self, request, *args, **kwargs):
        books = Book.objects.all()
        context = {'books': books}
        return self.render_to_response(context)
```

✅ Here, `render_to_response()` is inherited from `TemplateResponseMixin`.

render_to_response() can also be inherited from ListView and other CBVs