🧩 1. What are Django Decorators?

Decorators in Django (and Python) are functions that modify the behavior of other functions or methods — *without permanently changing their code.* You "wrap" another function or view with them.

✅ Example: Python concept

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before the function runs")
        result = func(*args, **kwargs)
        print("After the function runs")
        return result
    return wrapper

@my_decorator
def greet():
    print("Hello!")

greet()
```

Output:

```pgsql
Before the function runs
Hello!
After the function runs
```

2. Django Use Case for Decorators

In Django, decorators are mainly used for **function-based views (FBVs)**. They help in adding **authorization, caching, permissions, and input validations** cleanly.

Common django decorators:

1. **@login_required**
2. **@require_http_methods(["GET", "POST"])**
    a. **@require_GET**
    b. **@require_POST**
3. **@csrf_exempt**
4. **@cache_page(60 * 15)**
    a. Caches view output for 15 minutes

🛠️ 3. Writing a Custom Decorator in Django

Let's write a custom decorator that allows only users with a specific email domain (e.g., `@company.com`) to access a view.

Example:

```python
from django.http import HttpResponseForbidden

def company_email_required(view_func):
    def wrapper(request, *args, **kwargs):
        if request.user.is_authenticated and request.user.email.endswith('@company.com'):
            return view_func(request, *args, **kwargs)
        return HttpResponseForbidden("Access denied: Not a company email user.")
    return wrapper
```

Usage:

```python
@company_email_required
def dashboard(request):
    return render(request, 'dashboard.html')
```

What are Mixins?

Mixins are Python **classes** that you can “mix in” with other classes to add reusable functionality. They are used **with class-based views (CBVs)** instead of decorators.

Django Use Case for Mixins

In Django, mixins let you add reusable functionality to **CBVs** such as:

- Requiring authentication
- Permission checking
- Custom filtering logic
- Restricting allowed users, etc.

Common Mixins that are there in Django:

1. LoginRequiredMixin
2. PermissionRequiredMixin
3. SuccessMessageMixin
4. FormMixin
5. CreateView
6. UpdateView

✅ Example: Using `LoginRequiredMixin`

```python
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView
from .models import Employee

class EmployeeListView(LoginRequiredMixin, ListView):
    model = Employee
    template_name = 'employee_list.html'
    login_url = '/login/'  # Optional custom login URL
```

> Order matters — **mixins go before the view class** in inheritance.

1. But this **LoginRequiredMixin** works for inbuilt Django User Model.

Example of a custom mixin in Django:

🧠 6. Writing a Custom Mixin in Django

Let's write a mixin that restricts views to only staff users.

```python
from django.http import HttpResponseForbidden

class StaffRequiredMixin:
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_authenticated or not request.user.is_staff:
            return HttpResponseForbidden("You are not authorized to view this page.")
        return super().dispatch(request, *args, **kwargs)
```

Django uses **Method Resolution Order (MRO)** — it searches methods from **left to right**. If multiple mixins override the same method (like `dispatch`), the **leftmost one** takes priority.

Explanation with a Real world example:

✅ Our **custom** `User` **model** (not Django's built-in one)

✅ A **JWT token system** (without using Django's built-in authentication)

✅ Two **custom decorators**:

- `@token_required` → verifies token
- `@staff_required` → verifies if user in token is staff

✅ Two **equivalent mixins** for CBVs:

- `TokenRequiredMixin`
- `StaffRequiredMixin`

🧱 Step 1: Custom User Model

Let's define a simple user model manually — just with id, username, password, is_staff, etc.

`models.py`

```python
from django.db import models

class User(models.Model):
    username = models.CharField(max_length=100, unique=True)
    password = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    is_staff = models.BooleanField(default=False)

    def __str__(self):
        return self.username
```

🔐 Step 2: JWT Setup

We'll use **PyJWT** for token creation and verification.

> Install first if needed:
>
> ```bash
> pip install PyJWT
> ```

1. **Writing JWT token code in Utils.py file.**

`utils.py`

python

```python
import jwt
from datetime import datetime, timedelta

SECRET_KEY = 'your-secret-key'   # keep this secure in production

def generate_token(user):
    payload = {
        'user_id': user.id,
        'username': user.username,
        'is_staff': user.is_staff,
        'exp': datetime.utcnow() + timedelta(hours=1)  # 1 hour expiry
    }
    return jwt.encode(payload, SECRET_KEY, algorithm='HS256')

def decode_token(token):
    try:
        return jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
    except jwt.ExpiredSignatureError:
        return None
    except jwt.InvalidTokenError:
        return None
```

Example Login View (Token Creation)

python

```python
from django.http import JsonResponse
from .models import User
from .utils import generate_token

def login_view(request):
    username = request.POST.get('username')
    password = request.POST.get('password')

    try:
        user = User.objects.get(username=username, password=password)
    except User.DoesNotExist:
        return JsonResponse({'error': 'Invalid credentials'}, status=401)

    token = generate_token(user)
    return JsonResponse({'token': token})
```

🧩 Step 3: Custom Decorators

`decorators.py`

python

```python
from django.http import JsonResponse
from .models import User
from .utils import decode_token

def token_required(view_func):
    def wrapper(request, *args, **kwargs):
        token = request.headers.get('Authorization')
        if not token:
            return JsonResponse({'error': 'Token missing'}, status=401)

        data = decode_token(token)
        if not data:
            return JsonResponse({'error': 'Invalid or expired token'}, status=401)

        try:
            user = User.objects.get(id=data['user_id'])
        except User.DoesNotExist:
            return JsonResponse({'error': 'User not found'}, status=404)

        # Attach user to request for later use
        request.user = user
        return view_func(request, *args, **kwargs)
    return wrapper
```

```python
def staff_required(view_func):
    def wrapper(request, *args, **kwargs):
        # Must already have user set (so use after @token_required)
        if not hasattr(request, 'user'):
            return JsonResponse({'error': 'Authentication required'}, status=401)

        if not request.user.is_staff:
            return JsonResponse({'error': 'Staff access only'}, status=403)

        return view_func(request, *args, **kwargs)
    return wrapper
```

🧠 Step 4: Example Views (Function-Based)

`views.py`

python

```python
from django.http import JsonResponse
from .decorators import token_required, staff_required

@token_required
def user_dashboard(request):
    return JsonResponse({'message': f'Welcome {request.user.username}!'})

@token_required
@staff_required
def admin_panel(request):
    return JsonResponse({'message': f'Hello Admin {request.user.username}!'})
```

⚙️ Step 5: Equivalent Mixins for Class-Based Views

`mixins.py`

python

```python
from django.http import JsonResponse
from django.views import View
from .models import User
from .utils import decode_token

class TokenRequiredMixin(View):
    def dispatch(self, request, *args, **kwargs):
        token = request.headers.get('Authorization')
        if not token:
            return JsonResponse({'error': 'Token missing'}, status=401)

        data = decode_token(token)
        if not data:
            return JsonResponse({'error': 'Invalid or expired token'}, status=401)

        try:
            request.user = User.objects.get(id=data['user_id'])
        except User.DoesNotExist:
            return JsonResponse({'error': 'User not found'}, status=404)

        return super().dispatch(request, *args, **kwargs)
```

🧩 4. StaffRequiredMixin

Checks if the authenticated user is a staff member.

python Copy c

```python
# mixins.py (continued)
class StaffRequiredMixin(View):
    def dispatch(self, request, *args, **kwargs):
        user = getattr(request, 'user', None)
        if not user:
            return JsonResponse({'error': 'User not authenticated'}, status=401)

        if not user.is_staff:
            return JsonResponse({'error': 'Staff access required'}, status=403)

        return super().dispatch(request, *args, **kwargs)
```

✅ This one:

- Reads `request.user` (so you should use `TokenRequiredMixin` **before** this one if both are needed)
- Checks if `is_staff == True`

🧩 5. Usage Examples

(a) Use only token check:

python Copy code

```python
class ProfileView(TokenRequiredMixin, View):
    def get(self, request):
        return JsonResponse({'message': f'Welcome {request.user.username}'})
```

```python
class AdminPanelView(TokenRequiredMixin, StaffRequiredMixin, View):
    def get(self, request):
        return JsonResponse({'message': f'Hello Admin {request.user.username}'})
```

🧩 8. Converting Decorators to Mixins

If you already have a decorator and want to use it with a CBV, Django provides a utility:

python

```python
from django.utils.decorators import method_decorator
from django.contrib.auth.decorators import login_required
from django.views import View

@method_decorator(login_required, name='dispatch')
class MyView(View):
    def get(self, request):
        ...
```

Extra Information:

🧱 5. Why Django Mixins Use `super()`

Mixins are designed to be **cooperative** — meaning, each mixin calls `super()` instead of hardcoding a specific parent.

Example:

python Copy code

```python
class LoginRequiredMixin:
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponse("Unauthorized")
        return super().dispatch(request, *args, **kwargs)

class MyView(LoginRequiredMixin, View):
    def get(self, request):
        return HttpResponse("Hello user!")
```

Here:

- MRO = `[MyView, LoginRequiredMixin, View, object]`
- So `super().dispatch()` in `LoginRequiredMixin` → calls `View.dispatch()`

That's why we always use:

python Copy code

```python
return super().dispatch(request, *args, **kwargs)
```

Middlewares:

⚙️ 1. What is Middleware?

Middleware is a layer of logic that sits *between the request and response cycle* in Django.

Whenever a request comes in or a response goes out, middleware can intercept it and perform actions like:

- Processing, validating, or modifying requests/responses
- Handling authentication or security
- Managing sessions or CSRF protection
- Logging, throttling, caching, etc.

Think of middleware as a pipeline or a filter chain.

🔄 2. Request → Response Lifecycle in Django

When a client hits your Django app:

1. Django receives the HTTP **request**.
2. It passes through each **middleware** (top to bottom).
3. The **view** function or class is executed.
4. The **response** goes back through each middleware (bottom to top).
5. Django returns the final **HTTP response** to the client.

```
css

Request → [M1 → M2 → M3] → View → [M3 → M2 → M1] → Response
```

Each middleware can modify or stop the flow.

🧠 3. When Are Middlewares Used?

You use middleware when you want to apply **common functionality globally** across all requests/responses, like:

Use Case	Description
Authentication	Identify logged-in users from cookies or tokens
Security	CSRF, XSS, Clickjacking protection
Session management	Handle user sessions
Performance	Add caching or request timing
Logging	Log request details globally
API Monitoring	Track request/response count, errors, etc.
Maintenance mode	Temporarily disable site access

🧩 4. Built-in Django Middlewares (Common Ones)

Middleware	Purpose
`AuthenticationMiddleware`	Associates users with requests
`SessionMiddleware`	Manages session data
`CsrfViewMiddleware`	Adds CSRF protection
`CommonMiddleware`	Adds standard headers, URL redirects
`SecurityMiddleware`	Adds HTTPS and security headers
`MessageMiddleware`	Handles messages between requests
`LocaleMiddleware`	Handles localization/internationalization
`CacheMiddleware`	Adds caching at middleware level

All these are defined in your `settings.py`:

🧩 10. Middleware vs Mixin — Key Difference

Feature	Middleware	Mixin
Level	Global (applies to all requests)	Per-view (specific views only)
Purpose	Modify/handle request & response objects before/after views	Add reusable logic inside specific views
Execution Point	Runs *before and after* every view	Runs *within* a particular CBV
Scope	Framework-wide	View-specific
Example	Authentication middleware, Security middleware	LoginRequiredMixin, StaffRequiredMixin