# ⚙️ What is Middleware in Django?

**Middleware** is like a pipeline that processes requests and responses as they go in and out of your Django app.

Think of it as a **"layer" between the request and the view**, or between the view and the response.

**🔁 How It Works**

When a request comes in:

1. Django passes the request through each **middleware class**, one by one.
2. Then it hits the **view**.
3. The **response** from the view goes back **through the middleware (in reverse order)**.

It's like this:

Client -> Middleware 1 -> Middleware 2 -> View -> Middleware 2 -> Middleware 1 -> Response

**🧱 What Can Middleware Do?**

Middleware can:

* Authenticate users
* Add custom headers
* Handle CORS
* Log requests
* Redirect based on conditions
* Compress response content
* Block suspicious requests (like bots or IPs)
* Modify requests or responses

**🔧 Example of Built-in Django Middleware**

Here are some built-in ones from Django:

|  |
| --- |
| MIDDLEWARE = [  'django.middleware.security.SecurityMiddleware',  'django.contrib.sessions.middleware.SessionMiddleware',  'django.middleware.common.CommonMiddleware',  'django.middleware.csrf.CsrfViewMiddleware',  'django.contrib.auth.middleware.AuthenticationMiddleware',  'django.contrib.messages.middleware.MessageMiddleware',  ] |

Each one adds some functionality:

* CsrfViewMiddleware: Adds CSRF protection
* AuthenticationMiddleware: Associates users with requests
* SessionMiddleware: Manages user sessions

**✨ Creating Your Own Custom Middleware**

|  |
| --- |
| # myapp/middleware.py  class MyCustomMiddleware:  def \_\_init\_\_(self, get\_response):  self.get\_response = get\_response  def \_\_call\_\_(self, request):  # Code before the view (e.g., logging)  print("Before view")  response = self.get\_response(request)  # Code after the view (e.g., modifying response)  print("After view")  return response  Then register it in settings.py:  MIDDLEWARE = [  'myapp.middleware.MyCustomMiddleware',  ] |

**✅ Why Middleware is Useful**

* Keeps code **modular and clean**
* Adds global features **without modifying views**
* Perfect for **cross-cutting concerns** (like logging, authentication, rate limiting, etc.)

# 📡 What are Signals in Django?

**Signals** are Django's way of letting certain parts of your app know that something has happened, **without tightly coupling the code together**.

They follow the **Observer design pattern**:

When an event happens (like saving a model), a “signal” is sent out, and **listeners** (functions you write) can respond to that signal.

**🔔 Why Use Signals?**

* To perform an action **automatically after a specific event** (like user creation, saving a model, deleting data).
* Helps keep your **code modular** (separation of concerns).
* Used for things like:
  + Sending welcome emails after user registration
  + Creating user profiles automatically
  + Logging changes or deletions

**⚙️ How Signals Work**

1. **Sender**: The model or action that emits the signal.
2. **Receiver**: The function that gets called when the signal is triggered.
3. **Signal**: The event itself (like post\_save, pre\_delete, etc.)

**🧪 Common Built-in Signals**

| **Signal** | **Triggered When** |
| --- | --- |
| pre\_save | Before a model’s save() method is called |
| post\_save | After a model’s save() method is called |
| pre\_delete | Before a model’s delete() method is called |
| post\_delete | After a model’s delete() method is called |
| m2m\_changed | When a many-to-many field is modified |
| request\_started | When a request starts |
| request\_finished | When a request ends |
| user\_logged\_in | When a user logs in |

**✍️ Example: Create Profile After User is Created**

|  |
| --- |
| # myapp/signals.py  from django.db.models.signals import post\_save  from django.contrib.auth.models import User  from django.dispatch import receiver  from .models import Profile  @receiver(post\_save, sender=User)  def create\_user\_profile(sender, instance, created, \*\*kwargs):  if created:  Profile.objects.create(user=instance)  Then in your app’s apps.py:  # myapp/apps.py  from django.apps import AppConfig  class MyappConfig(AppConfig):  default\_auto\_field = 'django.db.models.BigAutoField'  name = 'myapp'  def ready(self):  import myapp.signals # important: register the signal |

**🔐 Tips:**

* Always import your signals in apps.py, not models.py, to avoid circular imports.
* Use the @receiver decorator to link signals to functions cleanly.
* Don’t overuse signals — sometimes it's better to handle logic explicitly in views or forms.

**💡 Real-Life Use Cases**

* Automatically creating related models (like Profile)
* Sending emails when events happen
* Logging or analytics tracking
* Soft deletes (flagging instead of deleting)
* Updating cache when models change

# 🛡️ Session Authentication vs Token Authentication

| **Feature** | **Session Authentication** | **Token Authentication** |
| --- | --- | --- |
| **Storage** | Session stored on **server** | Token stored on **client** |
| **Stateless?** | ❌ **Stateful** | ✅ **Stateless** |
| **Best For** | Traditional web apps with Django templates | APIs / mobile apps / SPAs (React, Flutter, etc.) |
| **Security** | Relies on cookies (usually) | Relies on headers (e.g., Authorization) |
| **Scalability** | Less scalable (server stores session data) | More scalable (no server-side session storage) |
| **Login Flow** | Login → Server creates session → Sends session ID via cookie | Login → Server sends token → Client stores token (in localStorage, etc.) |
| **Logout** | Server deletes session | Client deletes token (optional blacklist) |

**📦 1. Session Authentication (Django Default)**

* User logs in → Django creates a **session on the server**.
* A **session ID** is sent to the client via a **cookie**.
* On each request, the browser sends this session ID, and Django **authenticates the user** using the stored session.

🔐 **Pros:**

* Easy to implement in Django (default)
* Built-in middleware and user management

❌ **Cons:**

* Tied to browsers and cookies
* Not ideal for mobile or external APIs

**🪙 2. Token Authentication (Common in APIs)**

Used in **Django REST Framework (DRF)** or other modern APIs.

* User logs in → Server generates a **token** (like a JWT or random string).
* Client stores token (usually in localStorage or app memory).
* On each request, client sends:
* Authorization: Token <your\_token\_here>

🔐 **Pros:**

* Stateless = scalable
* Great for APIs, mobile apps, microservices
* Easy to expire/revoke tokens if needed

❌ **Cons:**

* You have to handle token storage securely on client side
* No built-in logout mechanism unless you blacklist tokens

**🛠️ How to Use in Django/DRF?**

**🧩 For Session Auth:**

|  |
| --- |
| Use Django’s default auth system (django.contrib.auth)  Add SessionAuthentication in DRF:  REST\_FRAMEWORK = {  'DEFAULT\_AUTHENTICATION\_CLASSES': [  'rest\_framework.authentication.SessionAuthentication',  ]  } |

**🔐 For Token Auth:**

|  |
| --- |
| # Install Django REST Framework:  pip install djangorestframework  # Add TokenAuth:  REST\_FRAMEWORK = {  'DEFAULT\_AUTHENTICATION\_CLASSES': [  'rest\_framework.authentication.TokenAuthentication',  ]  }  # Generate tokens:  from rest\_framework.authtoken.models import Token  user = User.objects.get(username='your\_user')  token = Token.objects.create(user=user) |

**✅ In Summary:**

* Use **session authentication** for traditional websites with Django templates.
* Use **token authentication** (or even **JWT**) for APIs, mobile apps, or SPA frontends.