

Django: "Baterías Incluidas"

Django es un framework web que abraza la filosofía *"batteries included"* — viene equipado con componentes listos para usar desde el primer momento. Estas "baterías" son las aplicaciones preinstaladas que cubren las tareas más comunes en el desarrollo web: panel de administración, autenticación de usuarios, manejo de sesiones, sistema de mensajes y gestión de archivos estáticos.

La gran ventaja de este enfoque es que te permite concentrarte en desarrollar la lógica de negocio específica de tu proyecto, en lugar de perder tiempo reinventando funcionalidades básicas que toda aplicación web moderna necesita.

¿Qué son las aplicaciones preinstaladas?

Las aplicaciones preinstaladas son módulos completamente funcionales que vienen activados por defecto en la configuración de Django. Se encuentran listados en la variable `INSTALLED_APPS` dentro del archivo `settings.py` de tu proyecto.

Estos módulos residen en el paquete `django.contrib` y proporcionan funcionalidades base de forma modular, permitiéndote activarlas o desactivarlas según las necesidades específicas de tu aplicación.

Esta arquitectura modular es uno de los pilares del diseño de Django, facilitando tanto el desarrollo rápido como el mantenimiento a largo plazo.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
]
```

Objetivos de Aprendizaje 🧠

01

Reconocer la utilidad

Identificar cómo las apps preinstaladas aceleran el desarrollo web profesional

02

Comprender el aporte

Entender qué funcionalidad específica ofrece cada aplicación al ciclo de desarrollo

03

Experimentar en práctica

Implementar las apps en un proyecto real: Bikeshop, nuestra tienda de bicicletas

04

Personalizar y explorar

Modificar comportamientos predeterminados y explorar modelos usando el ORM de Django



django.contrib.admin — Tu Panel de Control

La aplicación `admin` es una de las joyas de Django: proporciona una interfaz web completa para administrar los datos de tu aplicación sin necesidad de programar vistas personalizadas ni formularios HTML. Es perfecta para que personal no técnico pueda gestionar contenido de forma segura y eficiente.


Características principales:

- Operaciones CRUD completas sobre tus modelos
- Búsqueda y filtrado automático de registros
- Personalización mediante clases `ModelAdmin`
- Sistema de permisos integrado

```
from django.contrib import admin
from .models import Bicicleta

@admin.register(Bicicleta)
class BicicletaAdmin(admin.ModelAdmin):
    list_display = ('marca', 'modelo',
                  'precio', 'disponible')
```



 **Actividad práctica: Crea tu primer superusuario ejecutando `python manage.py createsuperuser` en la terminal**

InlineAdmin: Edición Avanzada

Los InlineAdmin permiten editar modelos relacionados directamente desde la vista del modelo principal, creando una experiencia de edición fluida y eficiente. Esta funcionalidad es especialmente útil cuando trabajas con relaciones uno-a-muchos.

```
class DetalleInline(admin.TabularInline):
    model = DetalleOrden
    extra = 1
    fields = ['producto', 'cantidad', 'precio']

@admin.register(Orden)
class OrdenAdmin(admin.ModelAdmin):
    inlines = [DetalleInline]
    list_display = ['numero', 'cliente', 'fecha']
```

Beneficio: Puedes editar una orden y todos sus detalles desde una única pantalla, sin necesidad de navegar entre múltiples páginas.



Existen dos tipos principales: **TabularInline** (tabla compacta) y **StackedInline** (formato apilado para campos más complejos).

django.contrib.auth — Seguridad y Control

El sistema de autenticación de Django es robusto y completo, proporcionando todo lo necesario para gestionar usuarios, contraseñas, grupos y permisos de forma segura. Es la base sobre la cual construyes el control de acceso en tu aplicación.



Autenticación

Login, logout, y verificación de credenciales con hash seguro de contraseñas



Permisos

Sistema granular de permisos por modelo y acción (agregar, cambiar, eliminar)



Grupos


Organiza usuarios en grupos para asignar permisos de forma eficiente

```
from django.contrib.auth.decorators import login_required
from django.contrib.auth.decorators import permission_required
```

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

```
@permission_required('bicicletas.add_bicicleta')
def agregar_bicicleta(request):
    # Solo usuarios con permiso específico
    pass
```



 **Práctica: Implementa login/logout en tu proyecto y prueba el decorador `@login_required` en una vista**

Personalización del Modelo User

Aunque el modelo `User` predeterminado de Django es funcional, muchas veces necesitas agregar campos adicionales como teléfono, RUT, fecha de nacimiento, o avatar. Django permite personalizar completamente el modelo de usuario.

⚠ Importante: Esta personalización debe hacerse al inicio del proyecto, antes de ejecutar las primeras migraciones. Cambiar el modelo de usuario después puede ser complicado.

```
from django.contrib.auth.models import AbstractUser
```

```
class UsuarioPersonalizado(AbstractUser):
```

```
    telefono = models.CharField(  
        max_length=15,  
        null=True,  
        blank=True
```

```
)
```

```
    rut = models.CharField(  
        max_length=12,  
        unique=True
```

```
)
```

```
# En settings.py
```

```
AUTH_USER_MODEL = 'miapp.UsuarioPersonalizado'
```



Con esta configuración, todos los formularios de registro y autenticación usarán automáticamente tu modelo personalizado.

django.contrib.contenttypes — Relaciones Flexibles



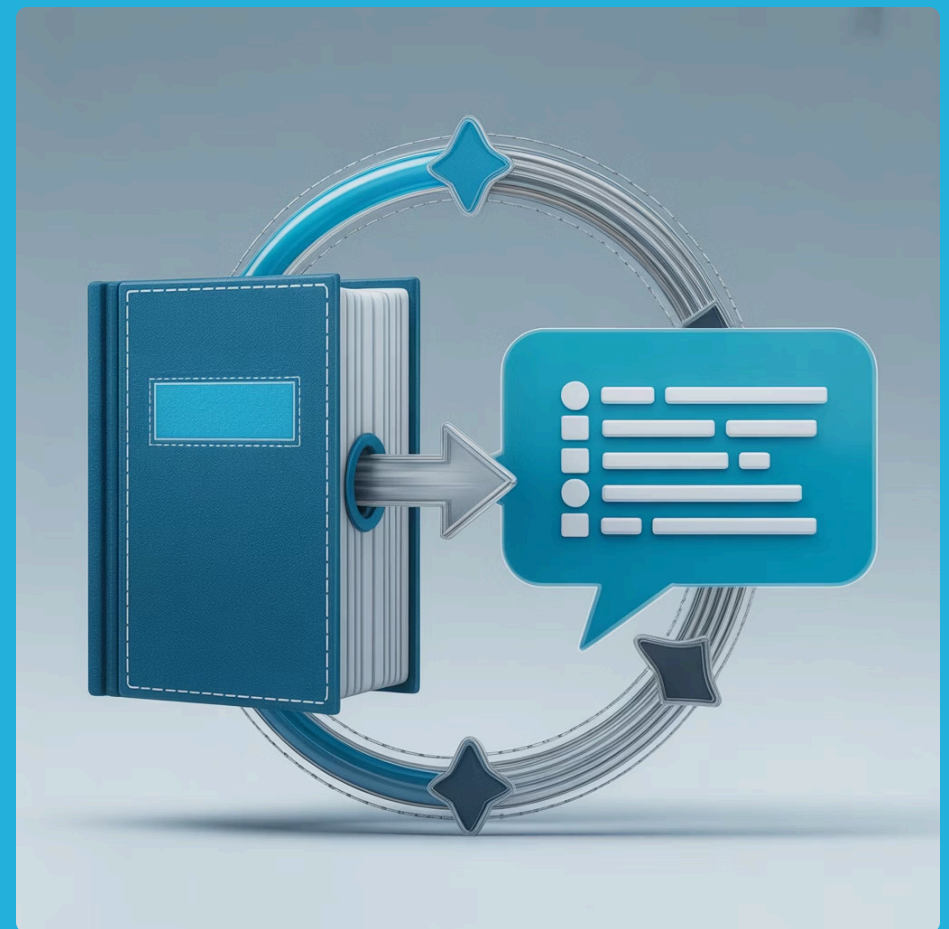
Esta aplicación permite crear relaciones genéricas: un modelo que puede relacionarse con cualquier otro modelo de tu aplicación. Es la base del sistema de permisos de Django y resulta útil para funcionalidades transversales como comentarios, etiquetas o favoritos.

```
from django.contrib.contenttypes.fields import (
    GenericForeignKey
)
from django.contrib.contenttypes.models import (
    ContentType

class Comentario(models.Model):
    # Campos que identifican el objeto relacionado
    content_type = models.ForeignKey(
        ContentType,
        on_delete=models.CASCADE
    )
    object_id = models.PositiveIntegerField()

    # Relación genérica
    content_object = GenericForeignKey(
        'content_type',
        'object_id'
    )

    texto = models.TextField()
    fecha = models.DateTimeField(auto_now_add=True)
```



Caso de uso: Un mismo modelo **Comentario** puede aplicarse a Bicicletas, Órdenes, Artículos del blog, o cualquier otro modelo sin duplicar código.

django.contrib.messages — Feedback Instantáneo



El framework de mensajes permite mostrar notificaciones temporales al usuario: mensajes de éxito, errores, advertencias o información. Son perfectos para proporcionar retroalimentación después de acciones como enviar formularios o completar operaciones.

En las vistas:

```
from django.contrib import messages

def crear_orden(request):
    if request.method == 'POST':
        # Procesar orden...
        messages.success(
            request,
            "¡Orden creada correctamente!"
        )
        messages.info(
            request,
            "Recibirás un email de confirmación"
        )
        return redirect('lista_ordenes')

    return render(request, 'crear_orden.html')
```

En los templates:

```
{% if messages %}
    {% for message in messages %}
        <div class="alert alert-{{ message.tags }}">
            {{ message }}
        </div>
    {% endfor %}
{% endif %}
```



Actividad: Agrega mensajes de confirmación al crear o editar una bicicleta en tu proyecto Bikeshop

django.contrib.sessions — Memoria Temporal

Las sesiones permiten almacenar información específica del usuario entre diferentes peticiones HTTP. Son fundamentales para implementar carritos de compra, preferencias del usuario, historial de navegación y cualquier dato temporal que no requiera persistencia en la base de datos.

Ejemplo: Carrito de Compras

```
def agregar_al_carrito(request, id):  
    # Obtener carrito actual o crear uno vacío  
    carrito = request.session.get('carrito', {})  
  
    # Incrementar cantidad del producto  
    carrito[str(id)] = carrito.get(str(id), 0) + 1  
  
    # Guardar en sesión  
    request.session['carrito'] = carrito  
    request.session.modified = True  
  
    messages.success(  
        request,  
        "Producto agregado al carrito"  
    )  
    return redirect('ver_carrito')  
  
def ver_carrito(request):  
    carrito = request.session.get('carrito', {})  
    total_items = sum(carrito.values())  
    # Mostrar carrito...
```



Configuración por defecto:

- Expiran después de 2 semanas de inactividad
- Se almacenan en la base de datos
- Identificadas por cookie segura

django.contrib.staticfiles — Recursos Estáticos

Esta aplicación gestiona todos los archivos estáticos de tu proyecto: hojas de estilo CSS, scripts JavaScript, imágenes, fuentes y otros recursos que no son generados dinámicamente. Proporciona un sistema organizado tanto para desarrollo como para producción.

En Desarrollo

Django sirve los archivos automáticamente desde las carpetas `static/` de cada aplicación

```
STATIC_URL = '/static/'
```

En Producción

Usa `collectstatic` para reunir todos los archivos en un único directorio que luego sirve un servidor web como Nginx



```
python manage.py collectstatic
```

Uso en templates:

```
{% load static %}
<link rel="stylesheet"
      href="{% static 'bicicletas/style.css' %}">

<script src="{% static 'js/carrito.js' %}"></script>
```



  **Práctica:** Crea tu propio archivo CSS en `static/bicicletas/style.css` y aplícalo a tus templates

Explorando con Django Shell

El shell interactivo de Django es una herramienta poderosa para explorar y probar código sin necesidad de crear vistas o templates. Te permite interactuar directamente con tus modelos, probar queries y experimentar con las APIs de Django.


Iniciar el shell:

```
python manage.py shell
```

Explorando el modelo User:

```
>>> from django.contrib.auth.models import User
>>>
>>> # Ver la tabla de base de datos
>>> print(User._meta.db_table)
auth_user
>>>
>>> # Listar todos los campos
>>> for field in User._meta.get_fields():
...     print(f'{field.name}: {field.get_internal_type()}')
>>>
>>> # Crear un usuario de prueba
>>> user = User.objects.create_user(
...     username='testuser',
...     email='test@example.com',
...     password='password123'
... )
>>>
>>> # Verificar usuarios existentes
>>> User.objects.all().count()
```



 **Ejercicio:** Usa el shell para explorar los campos del modelo `Session` y ver las sesiones activas en tu aplicación.

```
>>> from django.contrib.sessions.models import Session
>>> Session.objects.all()
```

Migraciones: Sincronizando la Base de Datos

Las migraciones son el mecanismo de Django para propagar cambios en tus modelos (agregar campos, crear tablas, etc.) a tu esquema de base de datos. Cada aplicación preinstalada necesita sus propias migraciones para crear las tablas necesarias.

1

Detectar cambios

```
python manage.py makemigrations
```

Analiza tus modelos y crea archivos de migración con los cambios detectados

2

Aplicar a BD

```
python manage.py migrate
```

Ejecuta las migraciones pendientes y actualiza el esquema de la base de datos

3

Verificar estado

```
python manage.py showmigrations
```

Muestra qué migraciones han sido aplicadas y cuáles están pendientes

Las apps preinstaladas como `auth`, `sessions` y `admin` crean automáticamente sus tablas la primera vez que ejecutas `migrate` en un proyecto nuevo.

Buenas Prácticas de Seguridad

Trabajar correctamente con las aplicaciones preinstaladas requiere seguir ciertas prácticas recomendadas para mantener tu aplicación segura, eficiente y mantenible.

django.contrib.admin

- Restringe acceso usando `is_staff` y `is_superuser`
- Cambia la URL por defecto `/admin/` a algo menos predecible
- Usa permisos granulares para diferentes tipos de usuarios
- Implementa autenticación de dos factores en producción

django.contrib.auth

- Nunca cambies `AUTH_USER_MODEL` después de migrar
- Usa contraseñas fuertes y validadores personalizados
- Implementa límites de intentos de login
- Siempre usa HTTPS en producción para proteger credenciales

django.contrib.sessions

- Evita almacenar datos sensibles en sesiones
- Configura `SESSION_COOKIE_SECURE = True` en producción
- Establece tiempos de expiración apropiados
- Limpia sesiones antiguas regularmente

django.contrib.messages

- Proporciona siempre feedback al usuario tras acciones importantes
- Usa niveles apropiados: `success`, `info`, `warning`, `error`
- No incluyas información sensible en mensajes

django.contrib.staticfiles

- Usa WhiteNoise o CDN en producción
- Minifica CSS y JavaScript antes de desplegar
- Implementa versionado de archivos para cache busting
- Nunca sirvas archivos estáticos con Django en producción

Caso Integrado: Proyecto Bikeshop






Nuestro proyecto Bikeshop demuestra cómo todas las aplicaciones preinstaladas trabajan juntas en un flujo completo de e-commerce. Cada componente cumple un rol específico en la experiencia del usuario.



Beneficios de la integración:

- Desarrollo rápido sin código repetitivo
- Seguridad robusta desde el primer día
- Experiencia de usuario fluida y profesional
- Fácil mantenimiento y escalabilidad

Apps utilizadas:

-  `admin` — gestión de productos
-  `auth` — login del personal
-  `sessions` — carrito temporal
-  `messages` — notificaciones
-  `staticfiles` — interfaz visual

Actividad Práctica 1: Explorando INSTALLED_APPS

Esta actividad te ayudará a comprender las dependencias internas entre las aplicaciones preinstaladas y cómo se afectan mutuamente.

01

Crear proyecto nuevo

Ejecuta `django-admin startproject experimento` para tener un entorno limpio de pruebas

02

Abrir settings.py

Localiza la sección `INSTALLED_APPS` y examina las aplicaciones listadas

03

Desactivar una app

Comenta temporalmente `'django.contrib.messages'` usando `#`

04

Ejecutar servidor

Intenta iniciar el servidor con `python manage.py runserver`

05

Observar errores

Analiza los mensajes de error en la terminal. ¿Qué otras apps dependen de messages?

  Reflexión: ¿Qué pasaría si desactivas `contenttypes`? ¿Por qué `auth` depende de ella?

Actividad Práctica 2: Admin y Autenticación

Trabaja en parejas para experimentar con el panel de administración y el sistema de autenticación. Un compañero será el administrador y el otro será el usuario regular.

Instrucciones para el Administrador:

1. Crear un superusuario con `createsuperuser`
2. Registrar el modelo `Bicicleta` en `admin.py`
3. Personalizar la vista con `list_display` y `list_filter`
4. Agregar al menos 5 bicicletas diferentes con datos realistas

```
@admin.register(Bicicleta)
class BicicletaAdmin(admin.ModelAdmin):
    list_display = ['marca', 'modelo', 'precio']
    list_filter = ['marca', 'disponible']
    search_fields = ['modelo', 'descripcion']
```

Instrucciones para el Usuario:

1. Crear una cuenta de usuario regular (sin permisos de staff)
2. Intentar acceder a `/admin/` — observar el comportamiento
3. Pedir al administrador que te otorgue permisos específicos
4. Acceder nuevamente y verificar qué puedes hacer



🎯 **Objetivo de aprendizaje:** Comprender la diferencia entre superusuarios, staff users y usuarios regulares, así como el sistema de permisos granulares de Django.

Actividad Práctica 3: Carrito en Sesión

Implementa un carrito de compras funcional usando el sistema de sesiones de Django. Esta actividad simula una característica real de e-commerce.

Paso 1: Crear la vista

```
from django.shortcuts import redirect, get_object_or_404
from django.contrib import messages
from .models import Bicicleta

def agregar_al_carrito(request, bicicleta_id):
    bicicleta = get_object_or_404(
        Bicicleta,
        id=bicicleta_id
    )

    carrito = request.session.get('carrito', {})
    bicicleta_key = str(bicicleta_id)

    if bicicleta_key in carrito:
        carrito[bicicleta_key]['cantidad'] += 1
    else:
        carrito[bicicleta_key] = {
            'nombre': bicicleta.modelo,
            'precio': float(bicicleta.precio),
            'cantidad': 1
        }

    request.session['carrito'] = carrito
    messages.success(
        request,
        f'{bicicleta.modelo} agregada al carrito'
    )

    return redirect('catalogo')
```

Paso 2: Vista del carrito

```
def ver_carrito(request):
    carrito = request.session.get('carrito', {})

    total = sum(
        item['precio'] * item['cantidad']
        for item in carrito.values()
    )

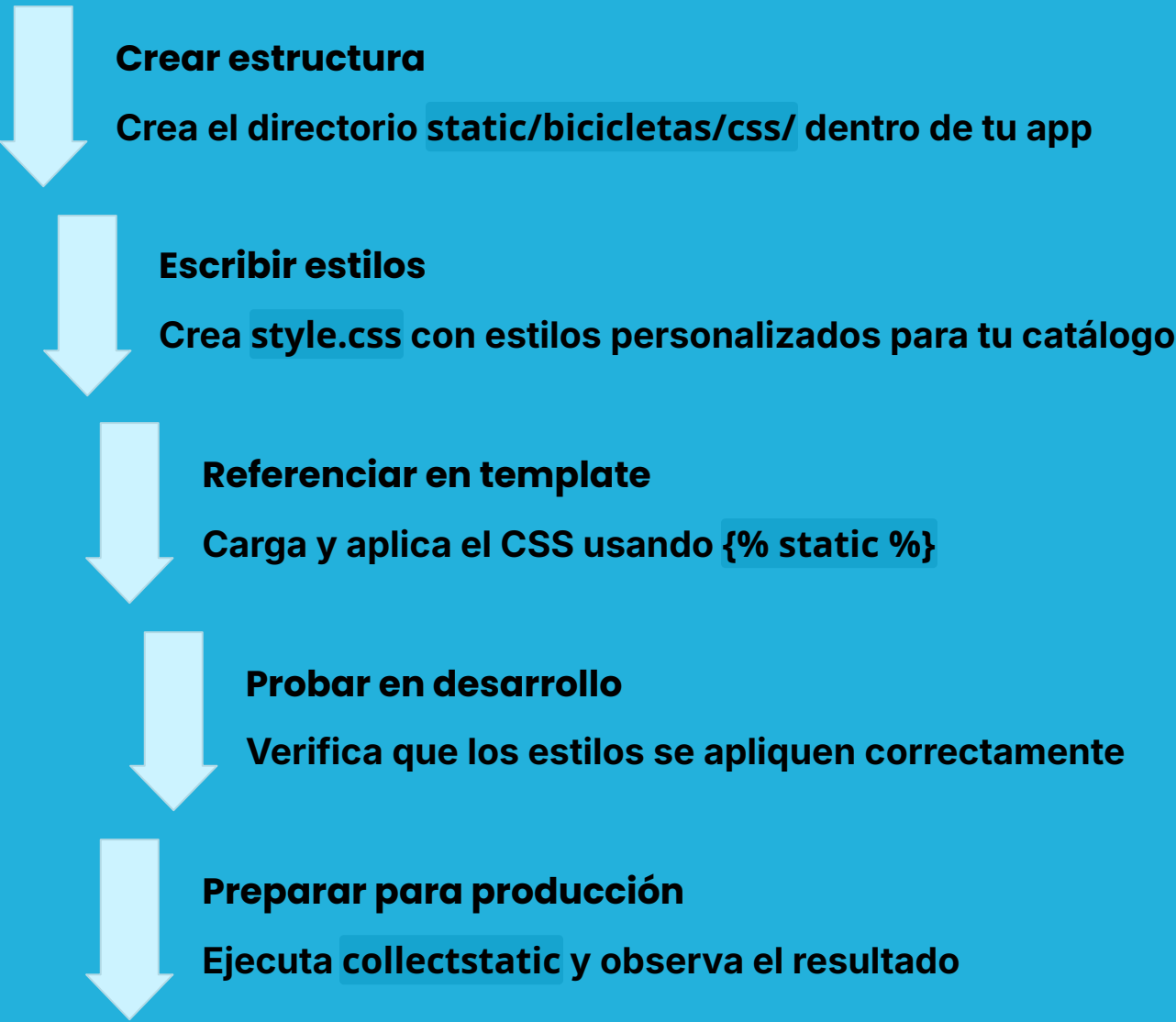
    total_items = sum(
        item['cantidad']
        for item in carrito.values()
    )

    return render(request, 'carrito.html', {
        'carrito': carrito,
        'total': total,
        'total_items': total_items
    })
```

🔗 Extensión: Agrega funcionalidad para incrementar/decrementar cantidades y eliminar productos del carrito. Muestra el número total de items en el header de todas las páginas.

Actividad Práctica 4: Archivos Estáticos

Personaliza la apariencia de tu aplicación Bikeshop creando y aplicando tus propios estilos CSS. Aprenderás a organizar archivos estáticos y referenciarlos correctamente en templates.



Ejemplo de CSS:

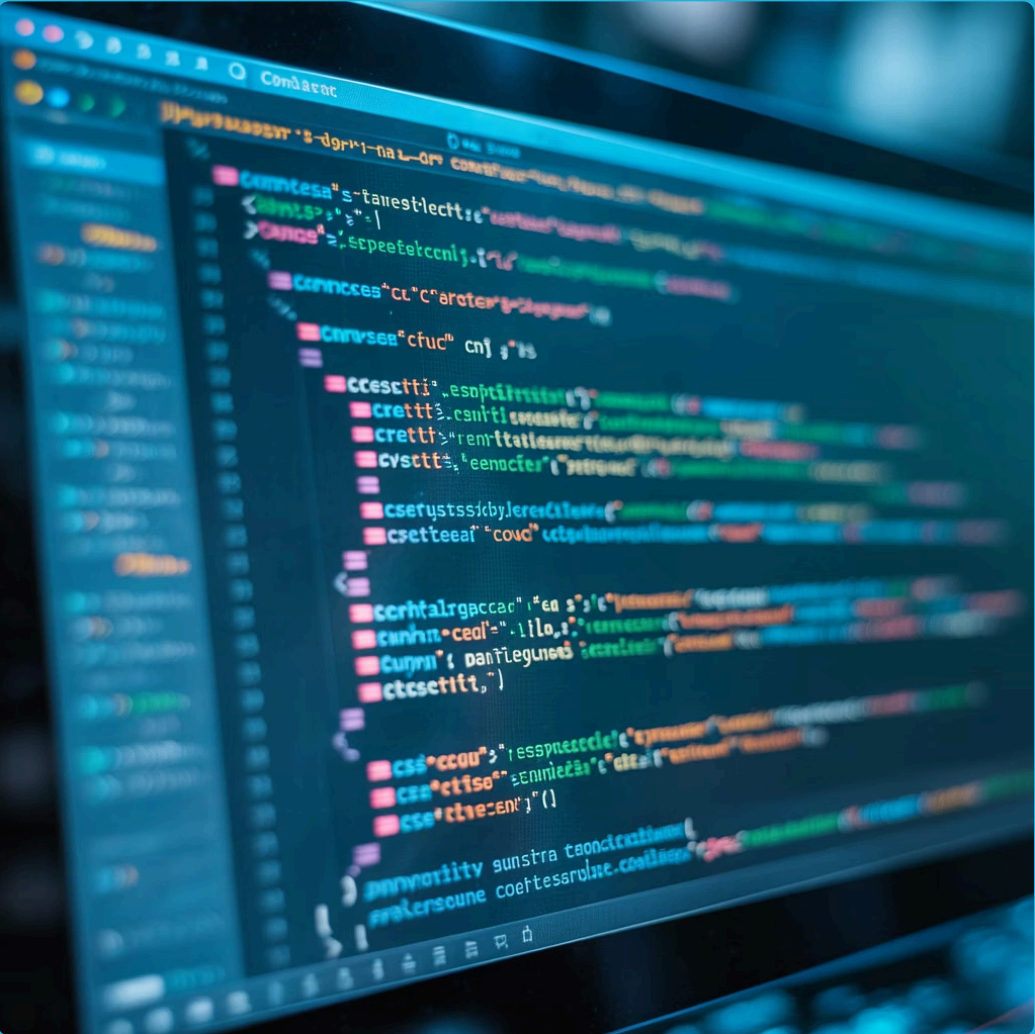
```
/* static/bicicletas/css/style.css */
.catalogo-grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 2rem;
  padding: 2rem;
}

.bicicleta-card {
  border: 1px solid #e0e0e0;
  border-radius: 8px;
  padding: 1.5rem;
  transition: transform 0.3s;
}

.bicicleta-card:hover {
  transform: translateY(-5px);
  box-shadow: 0 4px 12px rgba(0,0,0,0.1);
}
```

En el template:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet"
    href="{% static 'bicicletas/css/style.css' %}">
</head>
<body>
  <div class="catalogo-grid">
    {% for bici in bicicletas %}
      <div class="bicicleta-card">
        {{ bici.modelo }}
      </div>
    {% endfor %}
  </div>
</body>
</html>
```



Cierre: Las Herramientas de un Full Stack Developer



"No reinventes la rueda; aprende a usar bien las que ya giran"

Django trae herramientas integradas y probadas en producción que aceleran dramáticamente el desarrollo web profesional. Dominar estas aplicaciones preinstaladas es parte esencial del perfil de un Full Stack Python Developer.



Admin

Gestión de datos sin código adicional



Auth

Seguridad y permisos robustos



Sessions

Estado persistente entre peticiones



Messages

Feedback instantáneo al usuario



Staticfiles

Gestión eficiente de recursos



Reflexión final: Tómate un momento para compartir con el grupo: *¿Qué aplicación te pareció más útil y por qué? ¿Cómo la aplicarías en un proyecto personal?*

Estas herramientas no solo ahorran tiempo — te permiten construir aplicaciones profesionales y escalables desde el primer día. El verdadero poder está en saber combinarlas creativamente para resolver problemas reales.