



CRUD con Django – AE6

Bootcamp Full Stack Python · Módulo 7: Desarrollo de aplicaciones web con Django · Aprendizaje Esperado 6: "Implementar una aplicación web MVC que realiza operaciones CRUD utilizando Django"

Profesor: Cristian Iglesias · Proyecto base: bikeshop

Objetivos del Aprendizaje

Al finalizar esta clase, serás capaz de dominar los aspectos fundamentales del desarrollo web con Django, construyendo aplicaciones completas y funcionales.



Crear proyecto Django

Configurar un proyecto y aplicación Django desde cero para implementar operaciones CRUD completas



Operaciones CRUD

Implementar las cuatro operaciones básicas: Crear, Leer, Actualizar y Eliminar registros en la base de datos



Patrón MVT

Comprender y aplicar la interacción entre Model, View y Template en el flujo de datos de Django



Herramientas avanzadas

Dominar el uso de ModelForms, URLs dinámicas y CSRF Tokens para aplicaciones seguras

⚙️ ¿Qué es un CRUD?

CRUD es el acrónimo que define las cuatro operaciones básicas que se pueden realizar sobre cualquier base de datos. Este conjunto de operaciones forma la columna vertebral de prácticamente cualquier aplicación web moderna.

01

Create (Crear)

Agregar nuevos registros a la base de datos

02

Read (Leer)

Consultar y mostrar información existente

03

Update (Actualizar)

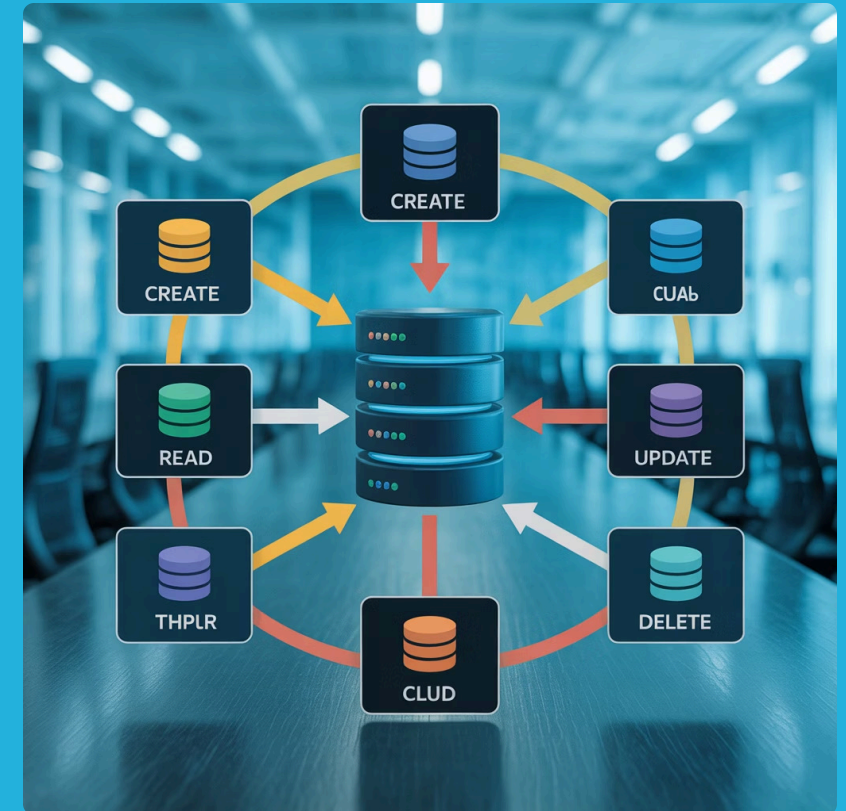
Modificar registros ya almacenados

04

Delete (Eliminar)

Remover registros de la base de datos

En Django, gestionamos estas operaciones a través del patrón MVT (Model-View-Template), que separa la lógica de datos, la lógica de negocio y la presentación visual.



💡 Actividad breve (3 min):

¿Dónde creen que ocurre cada operación dentro del patrón MVT de Django?



Recordemos la arquitectura MVT

El patrón MVT de Django es una variación del patrón MVC (Model-View-Controller) que separa las responsabilidades de una aplicación web en tres componentes interconectados. Esta arquitectura facilita el desarrollo, mantenimiento y escalabilidad de las aplicaciones.



Model (Modelo)

Define la estructura de datos y las reglas de negocio. Se comunica directamente con la base de datos mediante el ORM de Django, permitiendo crear, consultar, actualizar y eliminar registros sin escribir SQL directamente.



View (Vista)

Contiene la lógica que procesa las solicitudes HTTP del usuario. Recibe datos del modelo, los procesa según la lógica de negocio necesaria y decide qué template renderizar para mostrar la respuesta.



Template (Plantilla)

Renderiza los datos en formato HTML para presentarlos al usuario. Utiliza el lenguaje de plantillas de Django para insertar dinámicamente datos y lógica de presentación dentro del HTML.

Ejemplo práctico: El usuario solicita ver bicicletas → la vista intercepta la solicitud → pide los datos al modelo → el template recibe los datos y muestra la lista HTML → el navegador renderiza la página final.



Creación del Proyecto y Aplicación

Vamos a construir nuestro entorno de trabajo paso a paso. Primero crearemos el proyecto principal bikeshop y luego la aplicación bicicletas que contendrá toda la funcionalidad CRUD.

Paso 1: Crear el proyecto

```
django-admin startproject bikeshop  
cd bikeshop
```

Paso 2: Crear la aplicación

```
python manage.py startapp bicicletas
```

Paso 3: Verificar instalación

```
python manage.py runserver
```

Abre tu navegador en <http://127.0.0.1:8000/> y deberías ver la página de bienvenida de Django.

Paso 4: Registrar la app

En el archivo `settings.py`, agrega la aplicación a la lista de aplicaciones instaladas:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'bicicletas', # ← Nueva app  
]
```



Definiendo el Modelo Bicicleta

El modelo es la representación de la estructura de datos en nuestra aplicación. Define los campos que tendrá cada registro en la base de datos y sus tipos de datos.

Archivo: bicicletas/models.py

```
from django.db import models

class Bicicleta(models.Model):
    nombre = models.CharField(
        max_length=100
    )
    tipo = models.CharField(
        max_length=50
    )
    precio = models.DecimalField(
        max_digits=10,
        decimal_places=2
    )
    stock = models.PositiveIntegerField(
        default=0
    )
    created_at = models.DateTimeField(
        auto_now_add=True
    )

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

Análisis de campos

- **CharField:** Para textos cortos como nombre y tipo
- **DecimalField:** Para valores monetarios con precisión decimal
- **PositiveIntegerField:** Solo números enteros positivos para el stock
- **DateTimeField:** Registra automáticamente la fecha de creación
- **__str__:** Define cómo se representa el objeto en texto



👉 Actividad práctica: Escribe este modelo en VS Code y analiza cada campo con tu compañero. ¿Qué otros campos podrías agregar?

Migraciones: Sincronizando el Modelo con la Base de Datos

Las migraciones son el mecanismo de Django para propagar los cambios que realizas en tus modelos (agregar un campo, eliminar un modelo, etc.) hacia el esquema de tu base de datos. Cada vez que modificas un modelo, debes crear y aplicar migraciones.



Paso 1: Crear migración

```
python manage.py makemigrations  
bicicletas
```

Django detecta los cambios en `models.py` y genera un archivo de migración en la carpeta `migrations/`



Paso 2: Aplicar migración


```
python manage.py migrate
```

Django ejecuta las instrucciones SQL necesarias para crear o modificar las tablas en la base de datos



Resultado

Tu base de datos ahora tiene una tabla que refleja exactamente la estructura definida en tu modelo `Bicicleta`

 **Tip del profesor:** Las migraciones son como un control de versiones para tu base de datos. Mantienen sincronizado tu código Python con la estructura real de las tablas, y permiten trabajar en equipo sin conflictos en el esquema de datos.



Creando el Formulario con ModelForm

Los ModelForms son una de las características más poderosas de Django. Permiten generar automáticamente formularios HTML a partir de los modelos, ahorrando tiempo y garantizando consistencia entre el formulario y la base de datos.

Archivo: bicicletas/forms.py

```
from django import forms
from .models import Bicicleta

class BicicletaForm(forms.ModelForm):
    class Meta:
        model = Bicicleta
        fields = [
            'nombre',
            'tipo',
            'precio',
            'stock'
        ]
```

Este código simple genera automáticamente todos los campos del formulario con sus validaciones, tipos de entrada correctos y mensajes de error.

Ventajas de ModelForm

- **Generación automática:** Los campos se crean basándose en el modelo
- **Validación integrada:** Aplica las mismas reglas del modelo
- **Menos código:** No necesitas definir cada campo manualmente
- **Consistencia:** El formulario siempre refleja el modelo actual
- **Guardado simple:** `form.save()` crea o actualiza el registro directamente



Ejercicio práctico: Crea el archivo `forms.py`, escribe el código y ejecuta `python manage.py runserver`. En la siguiente diapositiva veremos cómo usarlo en una vista.



Configurando las URLs del CRUD

El sistema de enrutamiento de Django conecta las URLs que el usuario visita con las funciones de vista que procesan esas solicitudes. Necesitamos definir una ruta para cada operación CRUD.

Archivo: bicicletas/urls.py

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.lista_bicicletas, name='lista_bicicletas'),
    path('crear/', views.crear_bicicleta, name='crear_bicicleta'),
    path('actualizar/<int:pk>/', views.actualizar_bicicleta, name='actualizar_bicicleta'),
    path('eliminar/<int:pk>/', views.eliminar_bicicleta, name='eliminar_bicicleta'),
]
```

" (raíz)

Lista todas las bicicletas

URL: /bicicletas/

Vista: lista_bicicletas

'crear/'

Formulario para crear

URL: /bicicletas/crear/

Vista: crear_bicicleta

'actualizar/<int:pk>/'

Editar bicicleta existente

URL: /bicicletas/actualizar/5/

Vista: actualizar_bicicleta

'eliminar/<int:pk>/'

Eliminar una bicicleta

URL: /bicicletas/eliminar/5/

Vista: eliminar_bicicleta

No olvides incluir estas URLs en el archivo principal `bikeshop/urls.py`:

```
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('bicicletas/', include('bicicletas.urls')),
]
```



Vista: Crear Bicicleta

La vista de creación maneja dos escenarios: mostrar el formulario vacío cuando el usuario accede por primera vez (GET) y procesar los datos cuando el usuario envía el formulario (POST).

Archivo: bicicletas/views.py

```
from django.shortcuts import render, redirect
from .forms import BicicletaForm

def crear_bicicleta(request):
    if request.method == 'POST':
        form = BicicletaForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('lista_bicicletas')
    else:
        form = BicicletaForm()

    return render(
        request,
        'bicicletas/crear_bicicleta.html',
        {'form': form}
    )
```

Flujo de la vista

01

Usuario accede a /crear/ → Método GET → Se crea formulario vacío

02

Usuario llena formulario → Presiona "Guardar" → Método POST

03

Django valida datos → Si son válidos → Guarda en base de datos

04

Redirige a lista → Usuario ve la nueva bicicleta agregada



Actividad activa: Los estudiantes crean el template HTML correspondiente y prueban agregar una bicicleta desde el navegador. ¿Qué sucede si intentas enviar el formulario vacío?



Vista: Listar Bicicletas

La operación de lectura (Read) es generalmente la más simple del CRUD. Recupera todos los registros de la base de datos y los pasa al template para su visualización.

Archivo: bicicletas/views.py

```
from .models import Bicicleta

def lista_bicicletas(request):
    bicicletas = Bicicleta.objects.all()
    return render(
        request,
        'bicicletas/lista_bicicletas.html',
        {'bicicletas': bicicletas}
    )
```

¿Qué hace `objects.all()`?

El método `objects.all()` es parte del ORM de Django y ejecuta una consulta SQL equivalente a:

```
SELECT * FROM bicicletas_bicicleta;
```

Devuelve un QuerySet con todos los objetos Bicicleta almacenados.

Template: lista_bicicletas.html

```
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Tipo</th>
      <th>Precio</th>
      <th>Stock</th>
      <th>Acciones</th>
    </tr>
  </thead>
  <tbody>
    {% for b in bicicletas %}
      <tr>
        <td>{{ b.nombre }}</td>
        <td>{{ b.tipo }}</td>
        <td>${{ b.precio }}</td>
        <td>{{ b.stock }}</td>
        <td>
          <a href="{% url 'actualizar_bicicleta' b.pk %}">Editar</a>
          <a href="{% url 'eliminar_bicicleta' b.pk %}">Eliminar</a>
        </td>
      </tr>
    {% endfor %}
  </tbody>
</table>
```

👉 **Actividad:** Muestra en pantalla una tabla con todas las bicicletas. Prueba agregar algunas y observa cómo aparecen automáticamente en la lista.



Vista: Actualizar Bicicleta

La operación de actualización es similar a la de creación, pero primero debe recuperar el objeto existente de la base de datos y pre-llenar el formulario con sus datos actuales.

Archivo: bicicletas/views.py

```
from django.shortcuts import get_object_or_404

def actualizar_bicicleta(request, pk):
    bicicleta = get_object_or_404(Bicicleta, pk=pk)

    if request.method == 'POST':
        form = BicicletaForm(request.POST, instance=bicicleta)
        if form.is_valid():
            form.save()
            return redirect('lista_bicicletas')
    else:
        form = BicicletaForm(instance=bicicleta)

    return render(
        request,
        'bicicletas/actualizar_bicicleta.html',
        {'form': form, 'bicicleta': bicicleta}
    )
```

1

Recuperar objeto

`get_object_or_404` busca la bicicleta por su ID (pk). Si no existe, muestra un error 404 automáticamente en lugar de generar una excepción.

2

Pre-llenar formulario

El parámetro `instance=bicicleta` indica a `ModelForm` que debe mostrar los valores actuales del objeto en el formulario.

3

Guardar cambios

Cuando el usuario envía el formulario, `form.save()` actualiza el registro existente en lugar de crear uno nuevo.



Concepto clave: `get_object_or_404(Modelo, pk=id)` es una función de Django que evita errores inesperados. Si el objeto no existe, devuelve una página de error 404 amigable en lugar de romper la aplicación.



Vista: Eliminar Bicicleta

La operación de eliminación es la más directa del CRUD, pero requiere especial atención en términos de experiencia de usuario para evitar eliminaciones accidentales.

Archivo: bicicletas/views.py

```
def eliminar_bicicleta(request, pk):
    bicicleta = get_object_or_404(
        Bicicleta,
        pk=pk
    )
    bicicleta.delete()
    return redirect('lista_bicicletas')
```

Mejora recomendada

Para evitar eliminaciones accidentales, es mejor usar un formulario de confirmación:

```
def eliminar_bicicleta(request, pk):
    bicicleta = get_object_or_404(
        Bicicleta,
        pk=pk
    )

    if request.method == 'POST':
        bicicleta.delete()
        return redirect('lista_bicicletas')

    return render(
        request,
        'bicicletas/confirmar_eliminar.html',
        {'bicicleta': bicicleta}
    )
```

Template de confirmación

```
<h2>¿Confirmar eliminación?</h2>
<p>
    ¿Estás seguro de que deseas eliminar
    <strong>{{ bicicleta.nombre }}</strong>?
</p>
<p>Esta acción no se puede deshacer.</p>

<form method="POST">
    {% csrf_token %}
    <button type="submit">
        Sí, eliminar
    </button>
    <a href="{% url 'lista_bicicletas' %}">
        Cancelar
    </a>
</form>
```



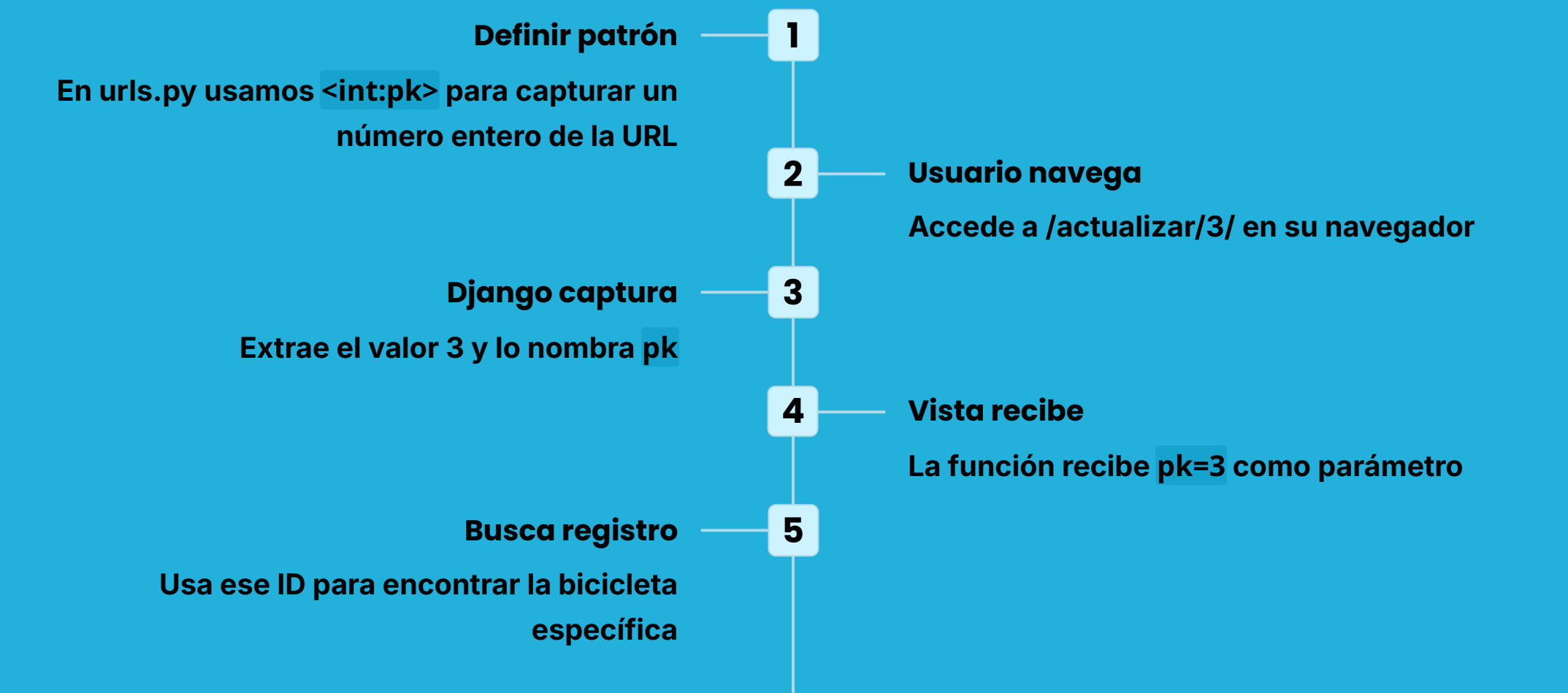
📄 🖱️ **Reflexión pedagógica:** Siempre debes confirmar acciones destructivas antes de ejecutarlas. Esto mejora la experiencia del usuario y previene pérdidas de datos accidentales.

🎯 **Reto breve:** Implementa una plantilla de confirmación que muestre los detalles de la bicicleta antes de eliminarla. ¿Qué otros datos mostrarías para ayudar al usuario a confirmar?



Enrutamiento y Parámetros Dinámicos

Las URLs dinámicas permiten que Django capture valores de la URL y los pase como argumentos a las vistas. Esto es esencial para operaciones CRUD que necesitan identificar registros específicos.



Tipos de parámetros en URLs

- `<int:pk>` - Captura números enteros (IDs)
- `<str:nombre>` - Captura cadenas de texto
- `<slug:titulo>` - Captura slugs (texto-con-guiones)
- `<uuid:id>` - Captura identificadores UUID

Ejemplo completo

```
path('actualizar/<int:pk>/',  
     views.actualizar_bicicleta,  
     name='actualizar_bicicleta')
```



📄 👉 **Actividad:** Cambia manualmente el número en la URL del navegador (por ejemplo, de `/actualizar/1/` a `/actualizar/5/`) y observa cómo cambia la bicicleta que se está editando.



Seguridad: Protección CSRF

Django incluye protección incorporada contra ataques CSRF (Cross-Site Request Forgery), un tipo de ataque donde sitios maliciosos intentan realizar acciones en tu aplicación usando la sesión autenticada del usuario.

¿Qué es CSRF?

Un ataque CSRF engaña al navegador del usuario para que envíe una solicitud no autorizada a tu sitio web. Por ejemplo, un sitio malicioso podría intentar eliminar datos o cambiar configuraciones sin que el usuario lo sepa.

¿Cómo funciona la protección?

1. Django genera un token único para cada sesión
2. El token se incluye en el formulario HTML
3. Cuando se envía el formulario, Django verifica el token
4. Si el token no coincide o falta, la solicitud es rechazada

Implementación en templates

Agrega `{% csrf_token %}` dentro de cada formulario que use método POST:

```
<form method="POST">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">
    Guardar
  </button>
</form>
```

Sin protección CSRF

```
<!--
```


Templates y Estructura de Diseño

La organización correcta de templates facilita el mantenimiento y la reutilización de código HTML. Django usa un sistema de plantillas potente que permite crear interfaces dinámicas de manera elegante.



Estructura recomendada

```
bicicletas/
├── templates/
│   └── bicicletas/
│       ├── base.html
│       ├── lista_bicicletas.html
│       ├── crear_bicicleta.html
│       ├── actualizar_bicicleta.html
│       └── confirmar_eliminar.html
```



Template base.html

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}BikeShop{% endblock %}</title>
</head>
<body>
    <nav>
        <a href="{% url 'lista_bicicletas' %}">Inicio</a>
        <a href="{% url 'crear_bicicleta' %}">Nueva
Bicicleta</a>
    </nav>
    {% block content %}
    {% endblock %}
</body>
</html>
```

Ejemplo de tabla con el lenguaje de templates

```
<table>
<thead>
<tr>
<th>Nombre</th>
<th>Tipo</th>
<th>Precio</th>
<th>Stock</th>
<th>Acciones</th>
</tr>
</thead>
<tbody>
{% for b in bicicletas %}
<tr>
<td>{{ b.nombre }}</td>
<td>{{ b.tipo }}</td>
<td>${{ b.precio }}</td>
<td>{{ b.stock }}</td>
<td>
<a href="{% url 'actualizar_bicicleta' b.pk %}">
```



Integrando MVT: El Flujo Completo

Ahora que hemos visto cada componente individualmente, veamos cómo trabajan juntos para crear una aplicación CRUD funcional. El flujo MVT es cíclico y cada componente tiene un rol específico.



Ejemplo concreto: Usuario hace clic en "Crear nueva bicicleta" → Django enruta a `crear_bicicleta` → La vista crea un formulario vacío → El template genera el HTML → Usuario completa el formulario → Django valida los datos → El modelo guarda en la base de datos → Redirige a la lista de bicicletas.



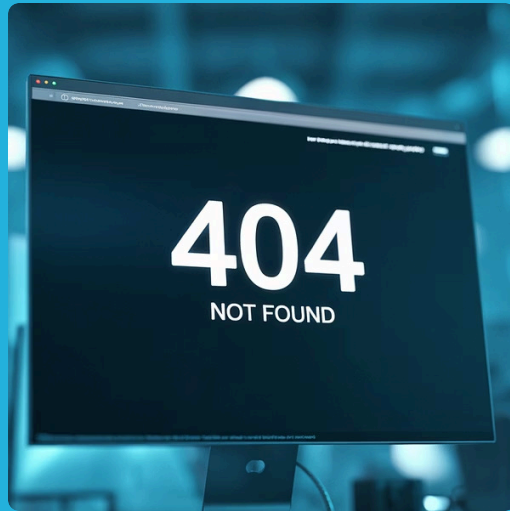
Buenas Prácticas en Django CRUD

Seguir buenas prácticas desde el principio te ahorrará horas de debugging y facilitará el mantenimiento de tu código. Estas recomendaciones se basan en la experiencia de miles de desarrolladores Django.



Validación siempre

Valida todos los formularios con `form.is_valid()` antes de guardar datos. Nunca confíes en los datos del usuario sin verificarlos primero.



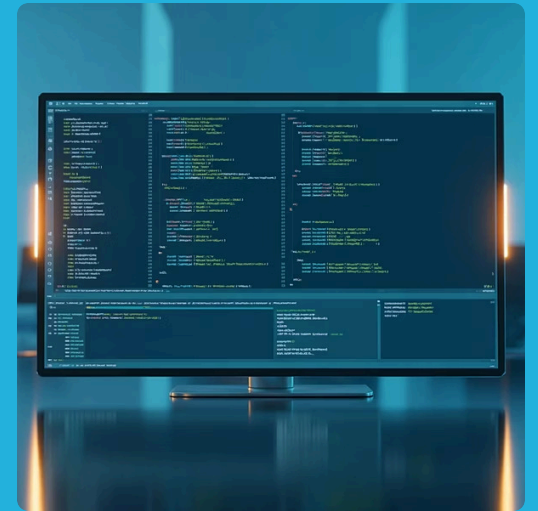
Manejo de errores

Usa `get_object_or_404()` en lugar de `.get()` para evitar excepciones no manejadas y mostrar páginas de error apropiadas.



Redirige después de POST

Siempre redirige después de un POST exitoso usando `redirect()`. Esto previene que el usuario re-envíe el formulario al recargar la página.



Nomenclatura consistente

Usa nombres descriptivos y consistentes para vistas, URLs y templates. Por ejemplo: `lista_bicicletas`, `crear_bicicleta`, `actualizar_bicicleta`.



Templates ordenados

Mantén la lógica fuera de los templates. Usa la vista para preparar los datos y el template solo para presentarlos. Aprovecha la herencia de templates.



Seguridad primero

Nunca olvides `{% csrf_token %}` en formularios POST. Valida permisos de usuario antes de realizar operaciones sensibles como eliminar.



Consejo del profesor: Estas prácticas pueden parecer trabajo extra al principio, pero te salvarán de errores difíciles de detectar más adelante. Un código limpio y seguro es un código que funciona bien a largo plazo.

Ejercicios Prácticos: ¡Manos a la Obra!

Ha llegado el momento de poner en práctica todo lo aprendido. Estos ejercicios te permitirán consolidar los conceptos y experimentar con el flujo completo CRUD en Django.

1 Crear registros

Accede a `/bicicletas/crear/` y agrega 5 bicicletas diferentes con distintos tipos, precios y cantidades de stock. Experimenta con valores válidos e inválidos para ver cómo funciona la validación.

2 Visualizar datos

Ve a la lista de bicicletas y verifica que todas se muestren correctamente en la tabla. ¿Se ordenan de alguna manera específica? ¿Cómo podrías ordenarlas por precio?

3 Modificar información


Selecciona una bicicleta y actualiza su precio. Cambia el stock de otra. Observa cómo el formulario se pre-llena con los datos actuales y cómo se reflejan los cambios en la lista.

4 Eliminar con cuidado

Elimina una de las bicicletas. Si implementaste la confirmación, verifica que funcione correctamente. ¿Qué sucede si intentas acceder a la URL de una bicicleta eliminada?

5 Prueba el flujo completo

Realiza el ciclo completo: crea una bicicleta nueva, edita sus datos, visualízala en la lista y finalmente elimínala. Observa cómo interactúan el navegador, las URLs, las vistas y los templates.

 **Actividad en parejas:** Trabajen juntos para completar estos ejercicios. Comparen resultados y ayúdense mutuamente a depurar errores. Discutan qué mejoras podrían agregar a la aplicación.

Bonus: Desafíos adicionales

- Agrega un campo "descripción" al modelo y actualiza el CRUD completo
- Implementa un buscador que filtre bicicletas por nombre o tipo
- Añade estilos CSS para mejorar la apariencia de las tablas y formularios
- Crea un mensaje de confirmación que aparezca después de cada operación exitosa

Cierre y Próximos Pasos

¿Qué hemos aprendido hoy?

🌟 Fundamentos del CRUD

Comprendimos las cuatro operaciones básicas sobre datos: Crear, Leer, Actualizar y Eliminar, y cómo implementarlas en Django de manera efectiva y segura.

🌟 Patrón MVT en acción

Vimos cómo el Modelo, la Vista y el Template trabajan juntos para crear una aplicación web completa, desde la base de datos hasta la interfaz del usuario.

🌟 Formularios dinámicos

Aprendimos a crear formularios seguros y funcionales usando ModelForms, que generan automáticamente campos validados basados en nuestros modelos.

🌟 Manipulación de datos web

Dominamos cómo manipular datos desde la interfaz web, incluyendo URLs dinámicas, validación de formularios y protección CSRF.

Desafío para casa

Ahora que dominas el CRUD de bicicletas, te propongo un reto: implementa un CRUD completo para otro modelo. Algunas ideas:

- **Cliente:** Con nombre, email, teléfono y dirección
- **Orden:** Relaciona clientes con bicicletas compradas
- **Repuesto:** Para piezas y accesorios de bicicletas
- **Empleado:** Para gestionar el personal de la tienda

Aplica todo lo aprendido: modelos, formularios, vistas, templates y buenas prácticas. ¿Te atreves a agregar relaciones entre modelos?

Recursos adicionales

- Documentación oficial de Django: [django-project.com](https://docs.djangoproject.com/)
- Django Girls Tutorial (en español)
- Real Python: Django Tutorials
- MDN Web Docs: Django

Próxima clase

En la siguiente sesión exploraremos cómo agregar relaciones entre modelos, autenticación de usuarios y vistas basadas en clases para hacer nuestras aplicaciones aún más potentes.

Recuerda: La práctica hace al maestro. Cuanto más construyas con Django, más natural se volverá el proceso. No tengas miedo de experimentar y cometer errores - son la mejor forma de aprender. ¡Éxito en tus proyectos! 🎉