

Django Templates: Separando Lógica de Presentación

Django permite separar la lógica de la presentación mediante templates, creando aplicaciones web más organizadas y mantenibles.

¿Qué son los Templates?

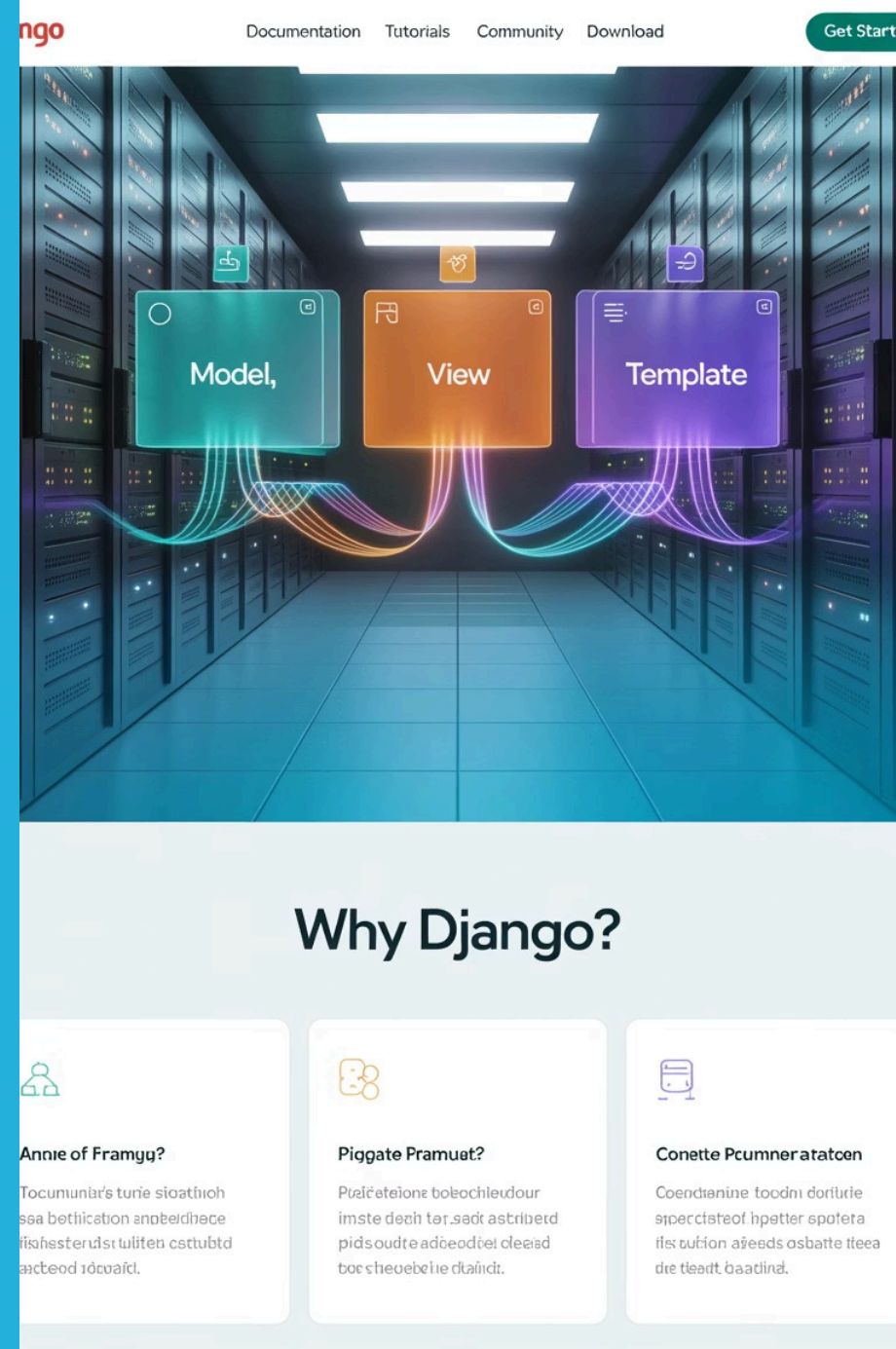
Archivos HTML con etiquetas especiales de Django que permiten mostrar datos dinámicos usando `{{ }}` y `{% %}`.

Objetivo Principal

Entender la función de los templates en la arquitectura Django y cómo facilitan el desarrollo web.

Ejercicio Activo

Abrir VS Code y crear `templates/index.html` con un elemento `<h1>` básico para comenzar.



Vistas en Django: El Puente entre Datos y Templates

Las vistas (views.py) gestionan la lógica de la aplicación y envían datos a los templates para su renderización. Son el componente central que conecta los modelos con las plantillas.

Ejemplo de Vista Básica

```
from django.shortcuts import render

def home(request):
    return render(request, 'index.html')
```

- ❏ **Ejercicio Activo:** Crear la vista home y mapearla correctamente en urls.py para establecer la ruta de acceso.



Navegador

Solicitud HTTP del usuario



Vista

Procesa lógica y datos



Template

Renderiza HTML dinámico



Respuesta

HTML final al navegador

Contexto y Variables: Enviando Datos a Templates

Las vistas pueden enviar datos al template mediante un diccionario de contexto, permitiendo que la información dinámica se muestre en las páginas web.

Código en la Vista

```
def perfil(request):
    usuario = {
        "nombre": "Carlos",
        "edad": 25
    }
    return render(request, 'perfil.html', usuario)
```

Uso en Template

```
Hola, {{ nombre }}
Tienes {{ edad }} años
```



Resultado: Hola, Carlos
Tienes 25 años

01 Definir Datos

Crear diccionario con información en la vista

03 Usar Variables

Acceder con {{ variable }} en el template

02 Pasar Contexto

Enviar datos como tercer parámetro en render()

04 Ejercicio

Modificar el nombre y refrescar la página para ver cambios



Plantillas Parciales: Componentes Reutilizables

Los componentes reutilizables como navbar, footer y sidebar permiten mantener un código más organizado y evitar la duplicación innecesaria.

Crear Componente

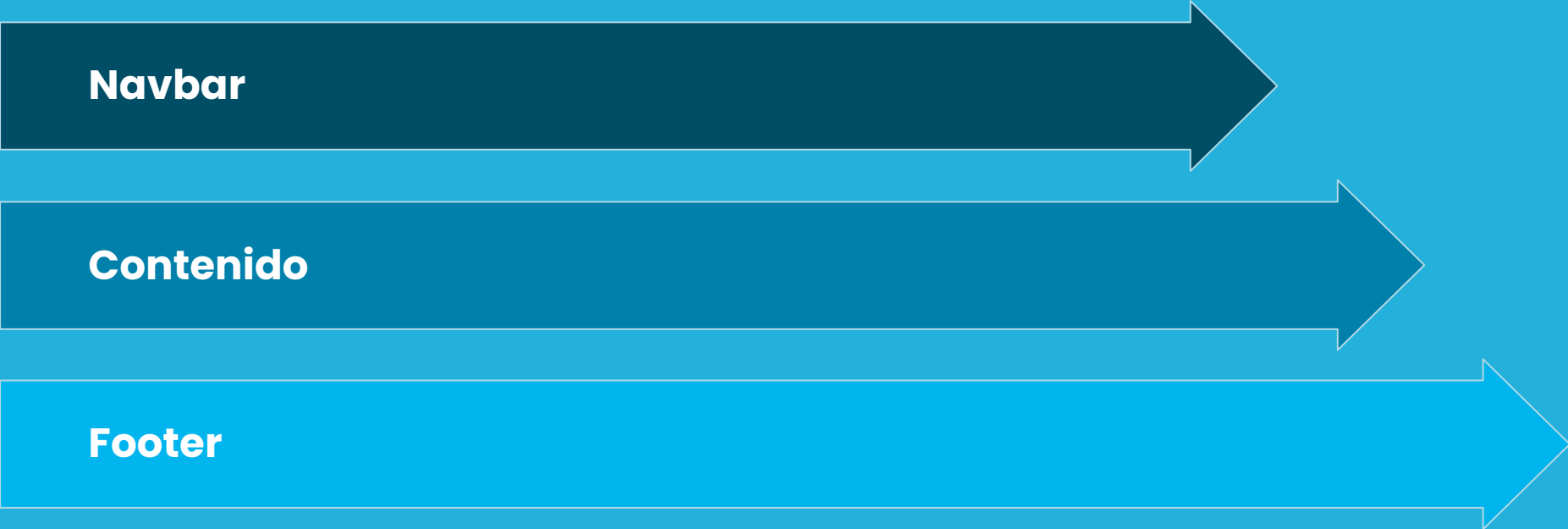
Desarrollar navbar.html como plantilla independiente con elementos de navegación comunes.

```
<nav>
  <ul>
    <li><a href="{% url 'home' %}">Inicio</a></li>
    <li><a href="{% url 'about' %}">Acerca</a></li>
  </ul>
</nav>
```

Incluir Componente

Usar `{% include 'navbar.html' %}` en otros templates para reutilizar el componente.

```
{% include 'navbar.html' %}
<main>
  <h1>Contenido principal</h1>
</main>
{% include 'footer.html' %}
```



Herencia de Plantillas: Estructura Jerárquica

Template Base (base.html)

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}Mi Sitio{% endblock %}</title>
</head>
<body>
  {% block content %}
  {% endblock %}
</body>
</html>
```

Template Hijo

```
{% extends 'base.html' %}

{% block title %}Página de Inicio{% endblock %}

{% block content %}
  <h1>Bienvenido</h1>
  <p>Contenido específico de la página</p>
{% endblock %}
```



📄 Ejercicio: Crear about.html extendiendo base.html con contenido personalizado para la página "Acerca de".

Crear Base

Definir estructura común con bloques `{% block %}` y `{% endblock %}`

Extender

Usar `{% extends 'base.html' %}` en plantillas hijas

Personalizar

Sobrescribir bloques específicos con contenido único

Iteradores en Templates: Recorriendo Listas

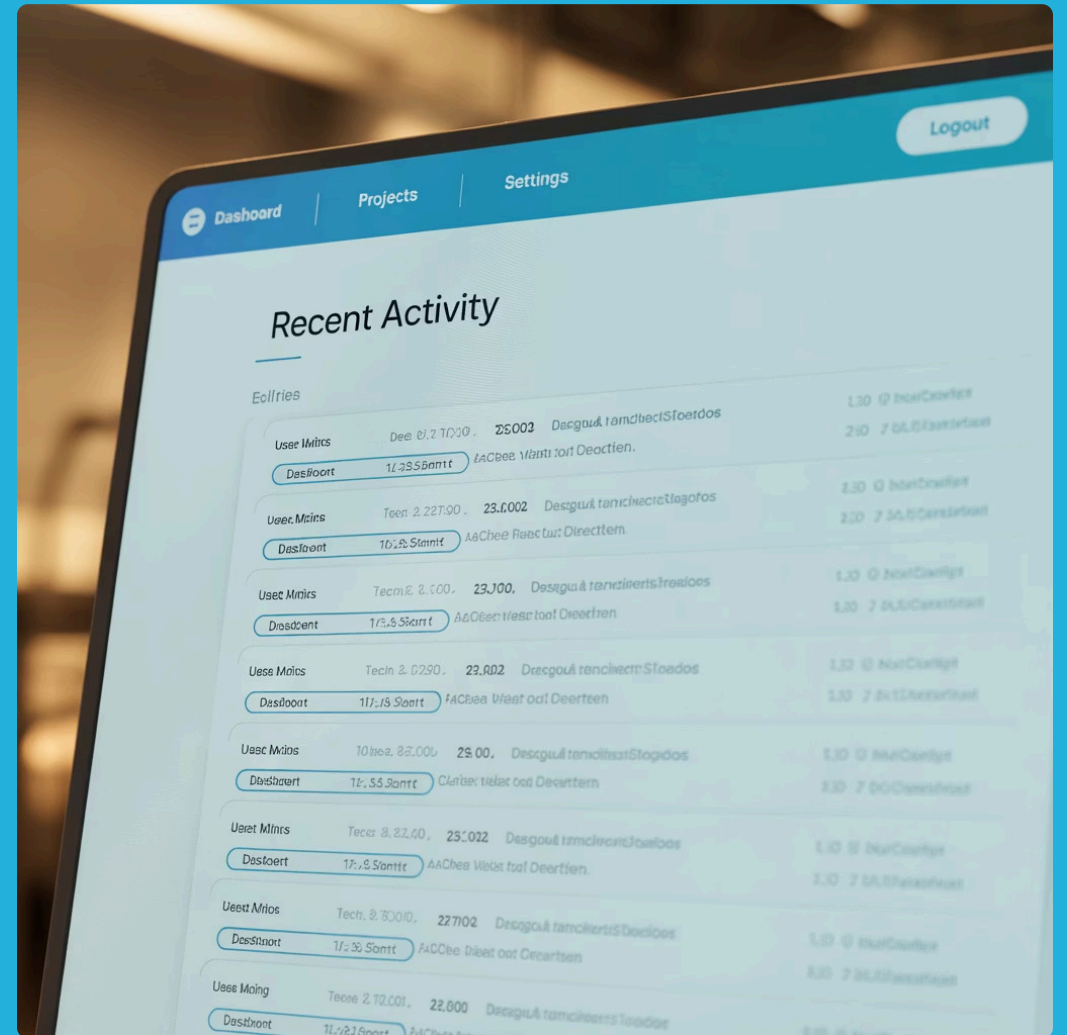
Los iteradores permiten recorrer listas y mostrar contenido dinámico de manera eficiente usando la etiqueta `{% for %}`.

Código en la Vista

```
def productos(request):  
    productos = [  
        {"nombre": "Laptop", "precio": 1200},  
        {"nombre": "Mouse", "precio": 25},  
        {"nombre": "Teclado", "precio": 80}  
    ]  
    return render(request, 'productos.html',  
                  {'productos': productos})
```

Template con Iterador

```
<ul>  
{% for producto in productos %}  
    <li>{{ producto.nombre }} - ${{ producto.precio }}</li>  
{% endfor %}  
</ul>
```



Resultado:

- Laptop - \$1200
- Mouse - \$25
- Teclado - \$80

📝 Ejercicio: Crear una lista de productos en `views.py` y mostrarla en el template usando iteradores para generar contenido dinámico.

Condiciones en Templates: Contenido Dinámico

Las condiciones permiten mostrar contenido diferente según el estado de variables usando `{% if %}`, `{% elif %}` y `{% else %}`.

Estructura Condicional

```
{% if usuario.is_authenticated %}
  <p>Bienvenido, {{ usuario.nombre }}</p>
  <a href="{% url 'logout' %}">Cerrar Sesión</a>
{% else %}
  <p>Inicia sesión para acceder</p>
  <a href="{% url 'login' %}">Iniciar Sesión</a>
{% endif %}
```

Condiciones Múltiples

```
{% if usuario.edad >= 18 %}
  <p>Acceso completo</p>
{% elif usuario.edad >= 13 %}
  <p>Acceso limitado</p>
{% else %}
  <p>Requiere supervisión</p>
{% endif %}
```

Usuario Autenticado

- ✓ Bienvenido, Carlos
- ✓ Cerrar Sesión


Usuario No Autenticado

- ⚠ Inicia sesión para acceder
- ⚠ Iniciar Sesión

📄 Ejercicio: Crear un usuario con atributo `is_authenticated` y probar diferentes condiciones para ver cómo cambia el contenido renderizado.

Filtros en Templates: Transformando Datos


Los filtros permiten transformar y formatear datos directamente en los templates sin modificar la lógica de las vistas.



Filtros de Texto

Transformar mayúsculas, minúsculas y capitalización:


```
{{ nombre | upper }}  
{{ nombre | lower }}  
{{ nombre | capfirst }}
```



Filtros Numéricos

Contar elementos y formatear números:

```
{{ productos | length }}  
{{ precio | floatformat:2 }}  
{{ fecha | date:"d/m/Y" }}
```



Combinación de Filtros


Aplicar múltiples transformaciones:

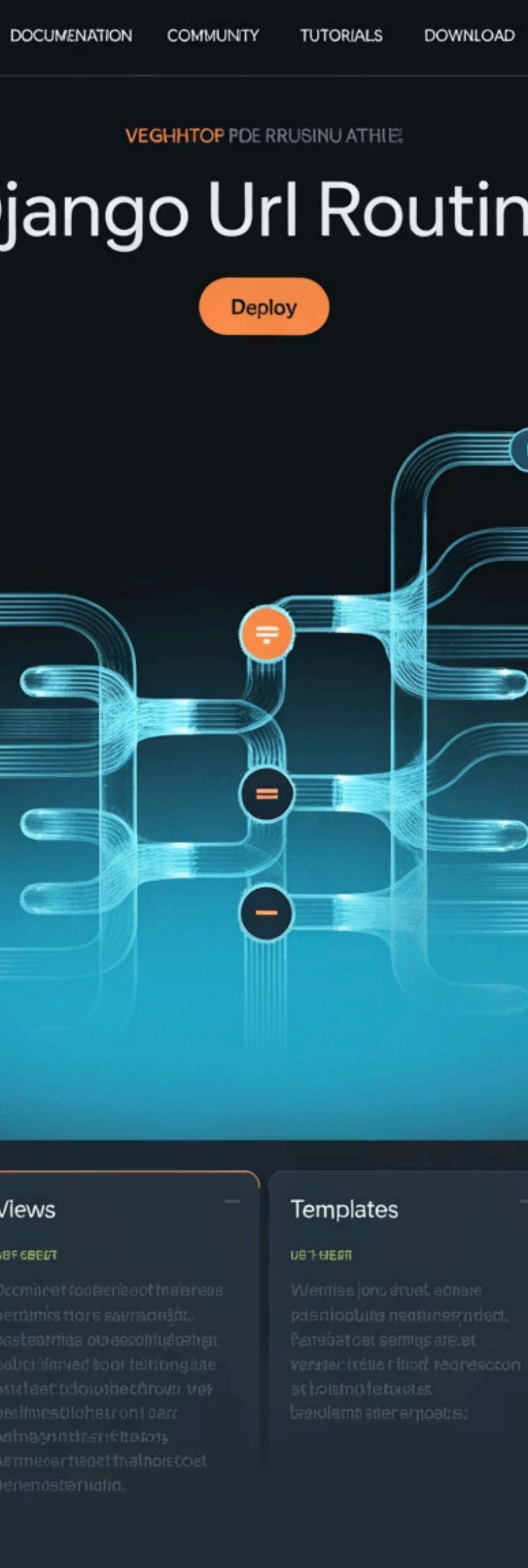
```
{{ nombre | lower | capfirst }}  
{{  
  descripcion | truncatewords:10  
}}  
{{ lista | slice:"3" }}
```

Ejemplo Práctico

```
{% for nombre in nombres %}  
  <p>{{ nombre | lower | capfirst }}</p>  
{% endfor %}
```



 **Ejercicio:** Combinar filtros aplicando `{{ nombre|lower|capfirst }}` a una lista de nombres para ver las transformaciones en acción.



Etiquetas URL: Enlaces Dinámicos

01

Definir URLs

Configurar rutas con nombres en urls.py para referenciarlas fácilmente

```
urlpatterns = [
    path("", views.home,
    name='home'),
    path('about/', views.about,
    name='about'),
]
```

02

Generar Enlaces

Usar {% url 'nombre_vista' %} para crear enlaces dinámicos

```
<a href="{% url 'home' %}">Inicio</a>
<a href="{% url 'about' %}">Acerca de</a>
```

03

Redirección en Vistas

Usar redirect('nombre_vista') para redirecciones programáticas

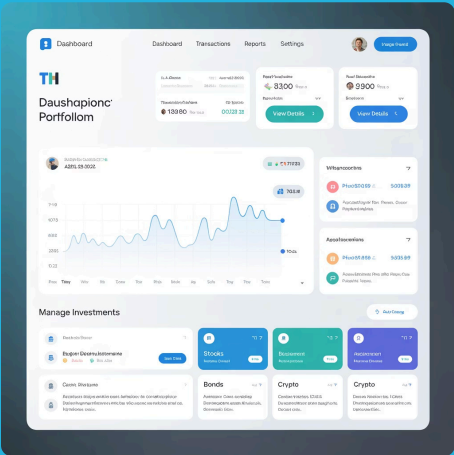
```
from django.shortcuts import redirect

def mi_vista(request):
    return redirect('home')
```

Ventajas de URLs Dinámicas

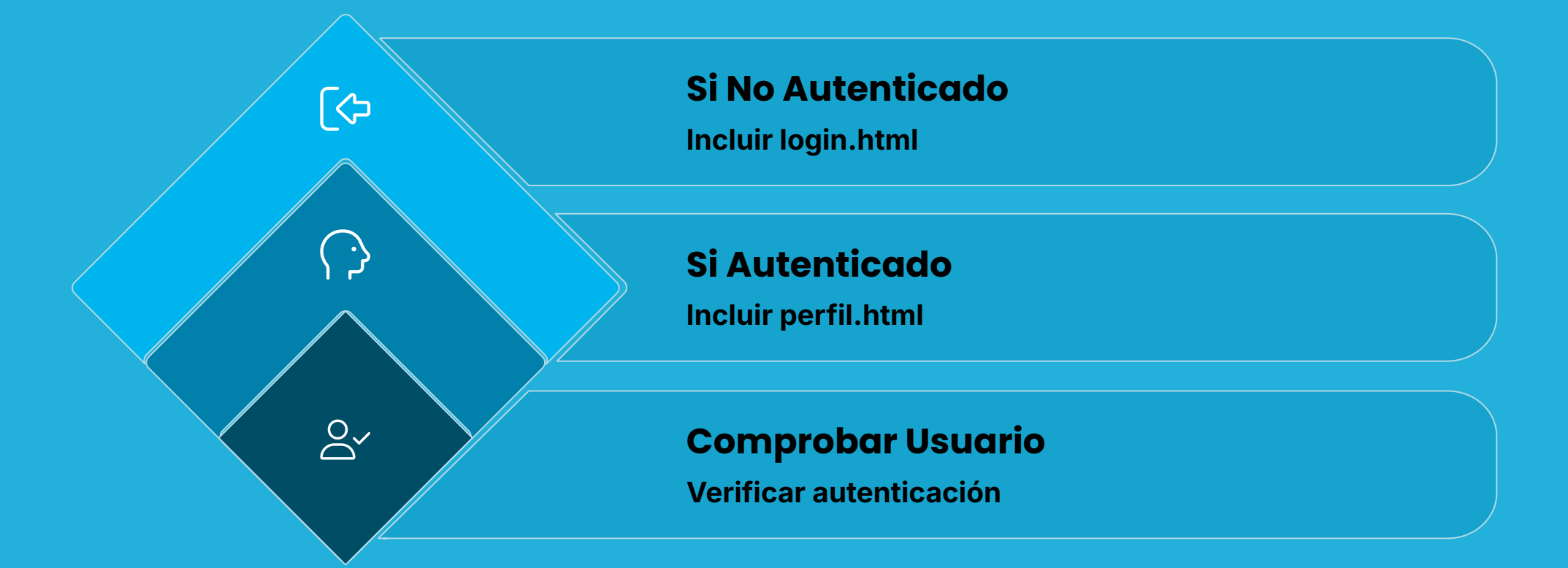
- Mantenimiento más fácil al cambiar rutas
- Enlaces siempre actualizados automáticamente
- Código más legible y mantenible
- Evita errores de enlaces rotos

Ejercicio: Crear un enlace a la página "about" usando {% url 'about' %} en lugar de URLs hardcodeadas.



Inclusión Condicional de Templates

La inclusión condicional permite mostrar diferentes templates según el estado de variables, creando experiencias personalizadas para los usuarios.



Bloques y Herencia Avanzada

La herencia avanzada permite sobrescribir bloques específicos de base.html, evitando duplicación de código y facilitando el mantenimiento de aplicaciones complejas.



Template Base Completo

Definir múltiples bloques para máxima flexibilidad:

```
{% block header %}{% endblock %}
{% block nav %}{% endblock %}
{% block content %}{% endblock %}
{% block sidebar %}{% endblock %}
{% block footer %}{% endblock %}
```



Sobrescritura Selectiva

Personalizar solo los bloques necesarios en cada página:

```
{% extends 'base.html' %}
{% block content %}
    <h1>Página Específica</h1>
{% endblock %}
{% block footer %}
    <p>Footer personalizado</p>
{% endblock %}
```

base.html

```
<header>
    {% block header %}
        <h1>Mi Sitio</h1>
    {% endblock %}
</header>
<main>
    {% block content %}{% endblock %}
</main>
<footer>
    {% block footer %}
        <p>© 2024 Mi Sitio</p>
    {% endblock %}
</footer>
```

about.html

```
{% extends 'base.html' %}

{% block content %}
    <h2>Acerca de Nosotros</h2>
    <p>Información de la empresa</p>
{% endblock %}

{% block footer %}
    <p>Contacto: info@empresa.com</p>
    <p>© 2024 Mi Empresa</p>
{% endblock %}
```

☐ Ejercicio: Crear un bloque footer personalizado en about.html que sobrescriba el footer por defecto de base.html.

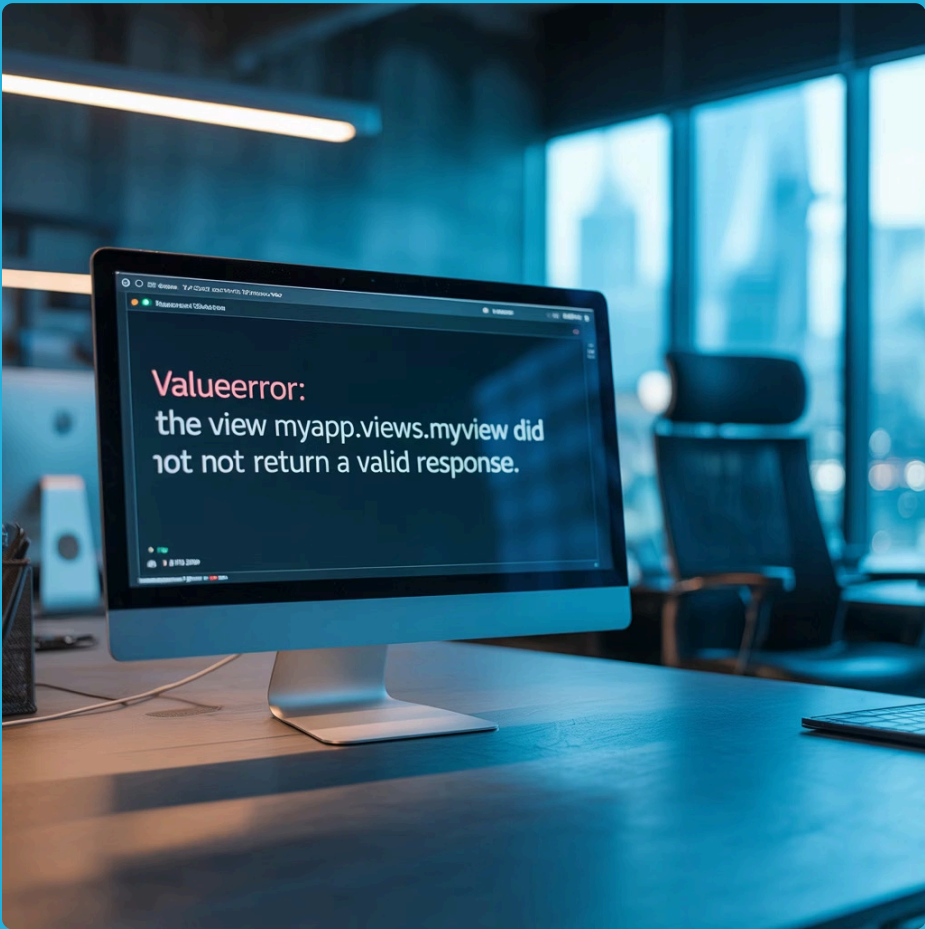
Manejo de Errores en Vistas

El manejo adecuado de errores en las vistas permite crear aplicaciones más robustas y proporcionar mejor experiencia al usuario.

Captura de Errores con try/except

```
def vista_calculadora(request):
    try:
        numero1 = int(request.GET.get('num1', 0))
        numero2 = int(request.GET.get('num2', 0))
        resultado = numero1 / numero2
        mensaje = f"Resultado: {resultado}"
    except ZeroDivisionError:
        resultado = None
        mensaje = "Error: No se puede dividir por cero"
    except ValueError:
        resultado = None
        mensaje = "Error: Valores no válidos"

    context = {
        'resultado': resultado,
        'mensaje': mensaje
    }
    return render(request, 'calculadora.html', context)
```



Template calculadora.html

```
<div class="resultado">
    {% if resultado %}
        <p class="success">{{ mensaje }}</p>
    {% else %}
        <p class="error">{{ mensaje }}</p>
    {% endif %}
</div>
```

01

Identificar Errores

Determinar qué errores pueden ocurrir en la vista

02

Implementar try/except

Capturar excepciones específicas con bloques try/except

03


Manejar Respuesta

Proporcionar mensajes informativos al usuario

04

Mostrar en Template

Renderizar mensajes de error de forma amigable

 **Ejercicio:** Simular un error de división por cero en la vista y mostrar un mensaje informativo en el template.

Uso de raise: Control de Excepciones

La declaración `raise` permite generar excepciones manualmente para controlar el flujo de la aplicación y validar datos de entrada.

1

Función de Validación

```
def verificar_edad(edad):  
    if edad < 0:  
        raise ValueError("La edad no puede ser negativa")  
    if edad < 18:  
        raise ValueError("Debe ser mayor de edad")  
    if edad > 120:  
        raise ValueError("Edad no válida")  
    return True
```

2

Uso en Vista

```
def registro(request):  
    try:  
        edad = int(request.POST.get('edad'))  
        verificar_edad(edad)  
        # Procesar registro exitoso  
        return render(request, 'exito.html')  
    except ValueError as e:  
        return render(request, 'registro.html',  
                        {'error': str(e)})
```

Tipos de Excepciones Comunes

- **ValueError:** Valores incorrectos
- **TypeError:** Tipos de datos incorrectos
- **KeyError:** Claves no encontradas
- **AttributeError:** Atributos inexistentes

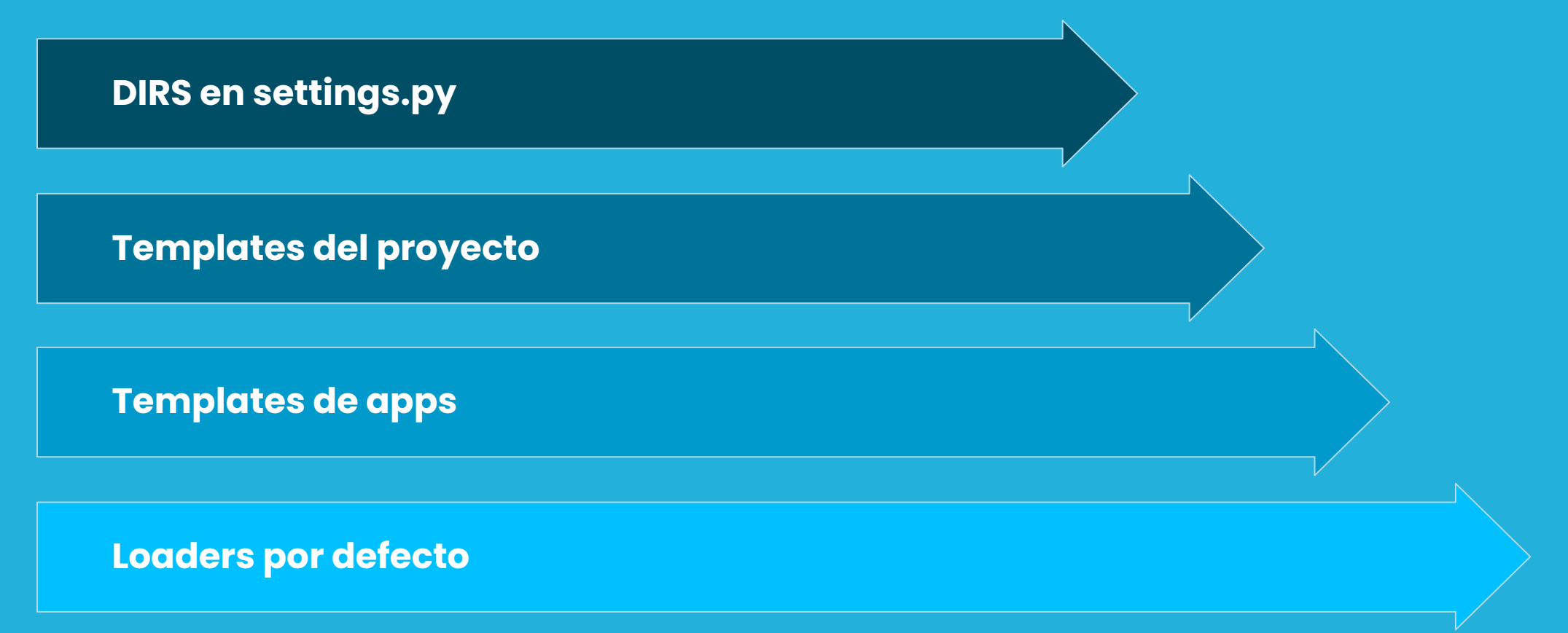
Prueba en Django Shell

```
python manage.py shell  
  
>>> from myapp.views import verificar_edad  
>>> verificar_edad(25) # True  
>>> verificar_edad(15) # ValueError  
>>> verificar_edad(-5) # ValueError
```

- 📄 **Ejercicio:** Probar la función `verificar_edad` con diferentes valores en la shell de Django para observar cómo se comportan las excepciones.

Jerarquía de Templates: Organización y Búsqueda

Django busca templates siguiendo una jerarquía específica: primero en DIRS de settings.py y luego en las aplicaciones instaladas.



Configuración en settings.py

```
TEMPLATES = [  
    {  
        'BACKEND':  
'django.template.backends.django.DjangoTemplates',  
        'DIRS': [BASE_DIR / 'templates'],  
        'APP_DIRS': True,  
        'OPTIONS': {  
            'context_processors': [  
                'django.template.context_processors.debug',  
                'django.template.context_processors.request',  
                'django.contrib.auth.context_processors.auth',  
            ],  
        },  
    },  
]
```

Estructura de Directorios

```
mi_proyecto/  
├── templates/      # Directorio global  
│   ├── base.html  
│   └── partials/  
│       ├── navbar.html  
│       └── footer.html  
└── mi_app/  
    ├── templates/  # Templates específicos de app  
    │   └── mi_app/  
    │       ├── index.html  
    │       └── detalle.html
```



Ejercicio: Crear una estructura de directorios organizada para templates globales y específicos de aplicación, incluyendo una carpeta `partials`.

Template con forloop.counter: Numeración Automática

La variable `forloop.counter` proporciona numeración automática en iteraciones, útil para crear listas numeradas y elementos indexados.

Código del Template

```
<div class="productos">
  {% for producto in productos %}
    <div class="producto-item">
      <h3>Producto {{ forloop.counter }}: {{ producto.nombre }}
    </h3>
      <p>Precio: ${{ producto.precio }}</p>
      <p>Posición: {{ forloop.counter }} de {{ forloop.revcounter
    }}</p>
    </div>
  {% endfor %}
</div>
```

Variables de forloop Disponibles

- `forloop.counter`: Índice desde 1
- `forloop.counter0`: Índice desde 0
- `forloop.revcounter`: Contador reverso
- `forloop.first`: Primer elemento
- `forloop.last`: Último elemento

Resultado Renderizado

Producto 1: Laptop

Precio: \$1200

Posición: 1 de 3

Producto 2: Mouse

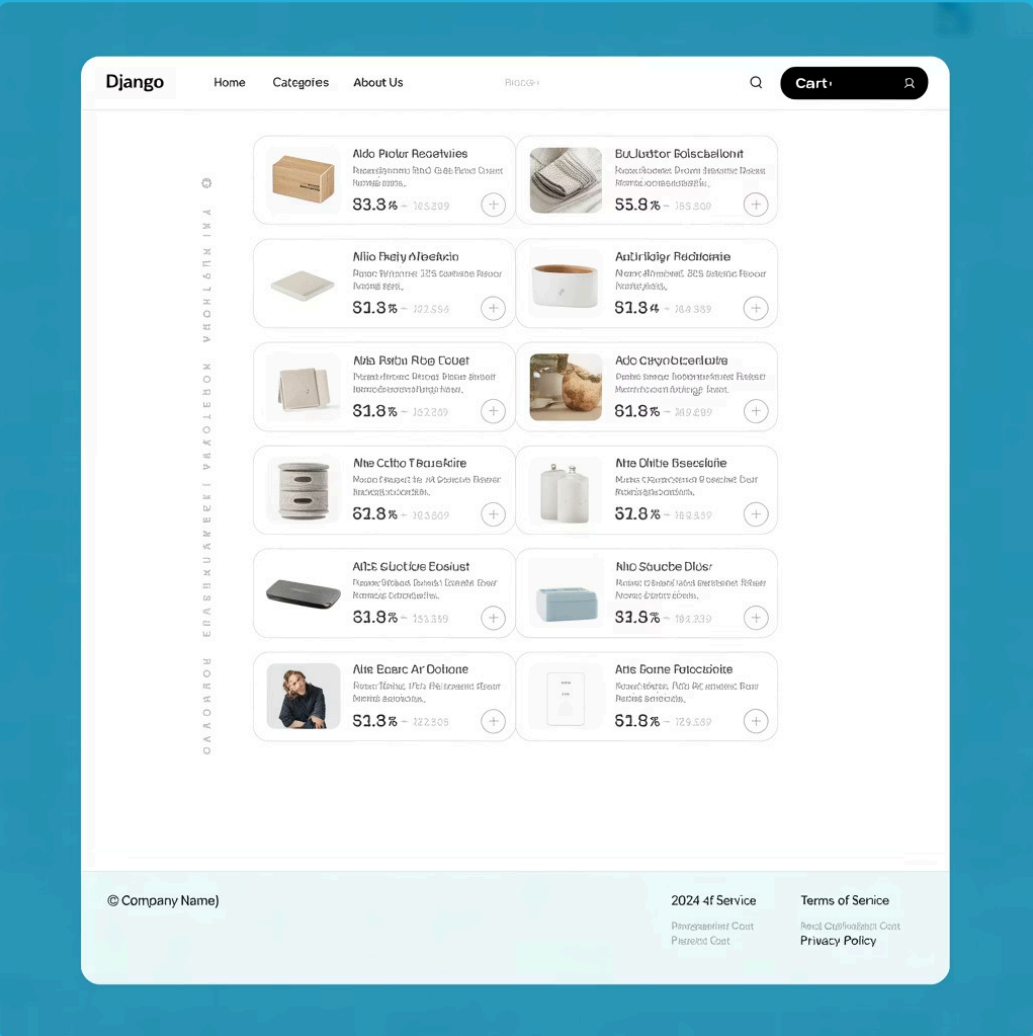
Precio: \$25

Posición: 2 de 2

Producto 3: Teclado

Precio: \$80

Posición: 3 de 1



Numeración Automática

No necesitas mantener contadores manuales, Django lo hace automáticamente

Estilos Condicionales

Usar `forloop.first` y `forloop.last` para aplicar estilos especiales

Información de Contexto

Proporcionar información sobre la posición del elemento en la lista

Ejercicio: Mostrar una lista de productos numerados usando `forloop.counter` para crear una interfaz más informativa.

Combinación de Filtros: Transformaciones Múltiples

La combinación de filtros permite aplicar múltiples transformaciones secuenciales a los datos, creando resultados más sofisticados y personalizados.

Filtros de Texto Encadenados

Aplicar múltiples transformaciones de texto:

```
{{ nombre | lower | capfirst | truncatechars: 20 }}
{{ descripcion | striptags | truncatewords: 5 }}
```

Filtros de Fecha y Hora

Formatear fechas con múltiples opciones:

```
{{ fecha | date:"d/m/Y" | default:"Sin fecha" }}
{{ timestamp | timesince | capfirst }}
```

Filtros Numéricos

Formatear números y cantidades:

```
{{ precio | floatformat:2 | add:"0" }}
{{ cantidad | pluralize:"producto,productos" }}
```

Ejemplo Práctico Completo

```
{% for producto in productos %}
  <div class="producto">
    <h3>{{ producto.nombre | upper | truncatechars:25 }}</h3>
    <p>{{ producto.descripcion | striptags | truncatewords:10 }}</p>
    <span class="precio">${{ producto.precio | floatformat:2 }}</span>
    <small>{{ producto.fecha | timesince }} atrás</small>
  </div>
{% endfor %}
```

Resultado de Transformaciones

Original: "laptop gaming profesional"	Filtrado: "LAPTOP GAMING PROFES..."
Original: 1299.999	Filtrado: "\$1300.00"
Original: 2024-01-15	Filtrado: "3 días atrás"

☐ Ejercicio: Aplicar filtros combinados {{ nombre|lower|capfirst }} a una lista de productos para ver las transformaciones secuenciales en acción.

extends y Modificación de Datos

Las plantillas hijas pueden sobrescribir bloques de la plantilla base, permitiendo personalización específica mientras mantienen la estructura común.

Definir Bloques Base

Crear estructura flexible en base.html con múltiples bloques personalizables

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}Mi Sitio{% endblock %}</title>
  {% block extra_css %}{% endblock %}
</head>
<body>
  {% block header %}
    <header><h1>Mi Sitio Web</h1></header>
  {% endblock %}

  {% block content %}{% endblock %}

  {% block footer %}
    <footer><p>© 2024</p></footer>
  {% endblock %}

  {% block extra_js %}{% endblock %}
</body>
</html>
```

Extender y Personalizar

Sobrescribir bloques específicos en plantillas hijas

```
{% extends 'base.html' %}

{% block title %}Contacto - Mi Sitio{% endblock %}

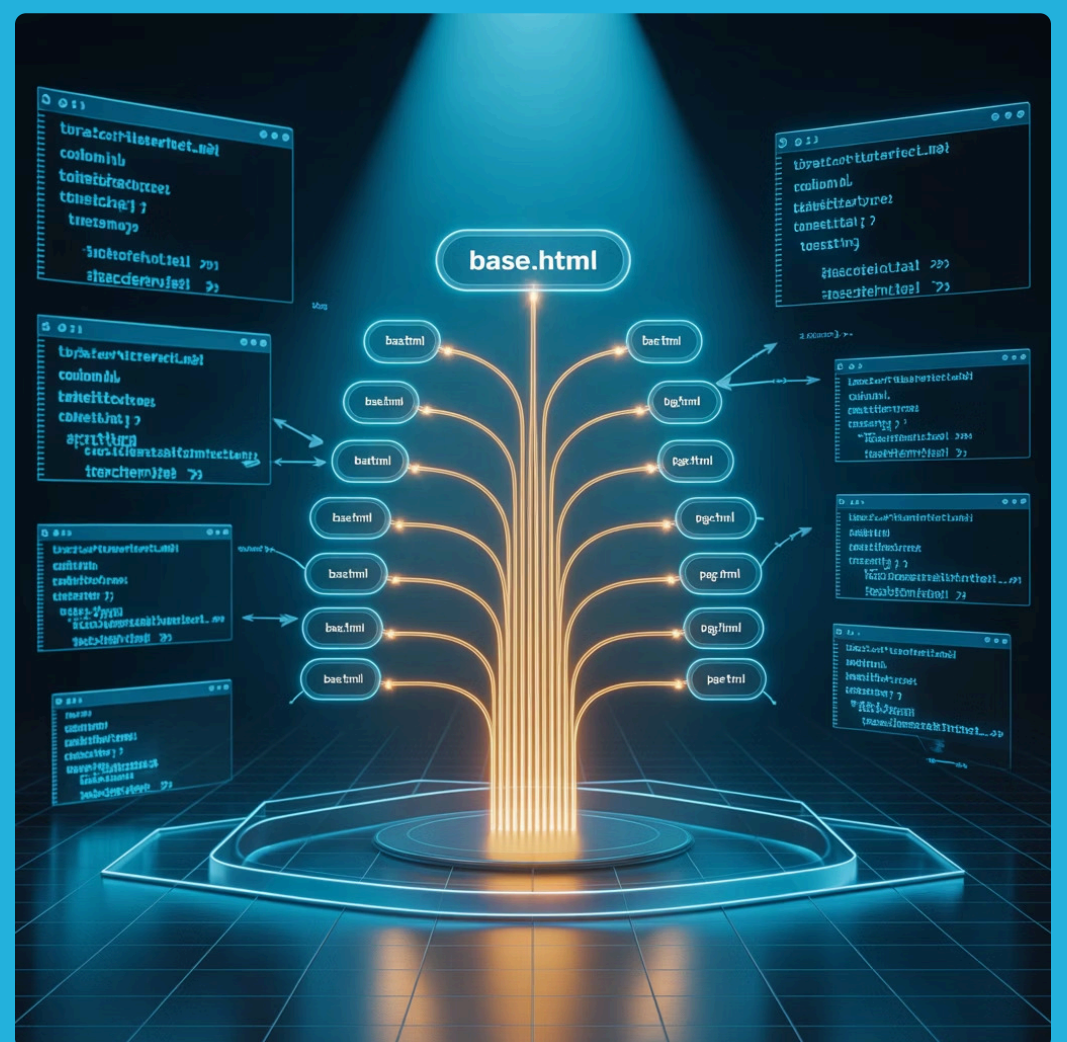
{% block extra_css %}
    <link rel="stylesheet" href="contacto.css">
{% endblock %}

{% block content %}
    <h2>Contáctanos</h2>
    <form method="post">
        <input type="text" name="nombre"
placeholder="Nombre">
        <textarea name="mensaje" placeholder="Mensaje">
    </textarea>
        <button type="submit">Enviar</button>
    </form>
{% endblock %}

{% block footer %}
    <footer>
        <p>Contacto: info@misitio.com</p>
        <p>© 2024 Mi Sitio Web</p>
    </footer>
{% endblock %}
```

Ventajas de la Herencia

- **Reutilización de código HTML común**
- **Mantenimiento centralizado de estructura**
- **Personalización flexible por página**
- **Consistencia visual automática**



- 📄 **Ejercicio: Crear una página de contacto extendiendo base.html con bloques personalizados para título, CSS y contenido específico.**

Importar Funciones en Templates

Django permite crear filtros y etiquetas personalizadas para extender la funcionalidad de los templates sin ensuciar el HTML con lógica compleja.

Crear Filtro Personalizado

En `mi_app/templatetags/mi_filtros.py`:

```
from django import template

register = template.Library()

@register.filter
def multiplicar_por_dos(valor):
    try:
        return int(valor) * 2
    except (ValueError, TypeError):
        return 0

@register.filter
def formatear_precio(precio):
    return f"${precio:,.2f}"
```

Usar en Template

```
{% load mi_filtros %}

<div class="productos">
{% for producto in productos %}
    <div class="producto">
        <h3>{{ producto.nombre }}</h3>
        <p>Precio: {{ producto.precio|formatear_precio }}</p>
        <p>Doble: {{ producto.precio|multiplicar_por_dos }}</p>
    </div>
{% endfor %}
</div>
```

01

Crear Directorio

Crear `mi_app/templatetags/` con `__init__.py`

02

Definir Filtros

Crear archivo con funciones decoradas con `@register.filter`

03

Cargar en Template

Usar `{% load mi_filtros %}` al inicio del template

04

Aplicar Filtros

Usar `{{ variable|mi_filtro }}` en el template

📌 **Ejercicio:** Crear un filtro personalizado `multiplicar_por_dos` y usarlo en un template para transformar valores numéricos.

Actividad Práctica Grupal: Mini-Tienda

Crear un proyecto completo de mini-tienda que integre todos los conceptos aprendidos: herencia, iteradores, filtros y condiciones.

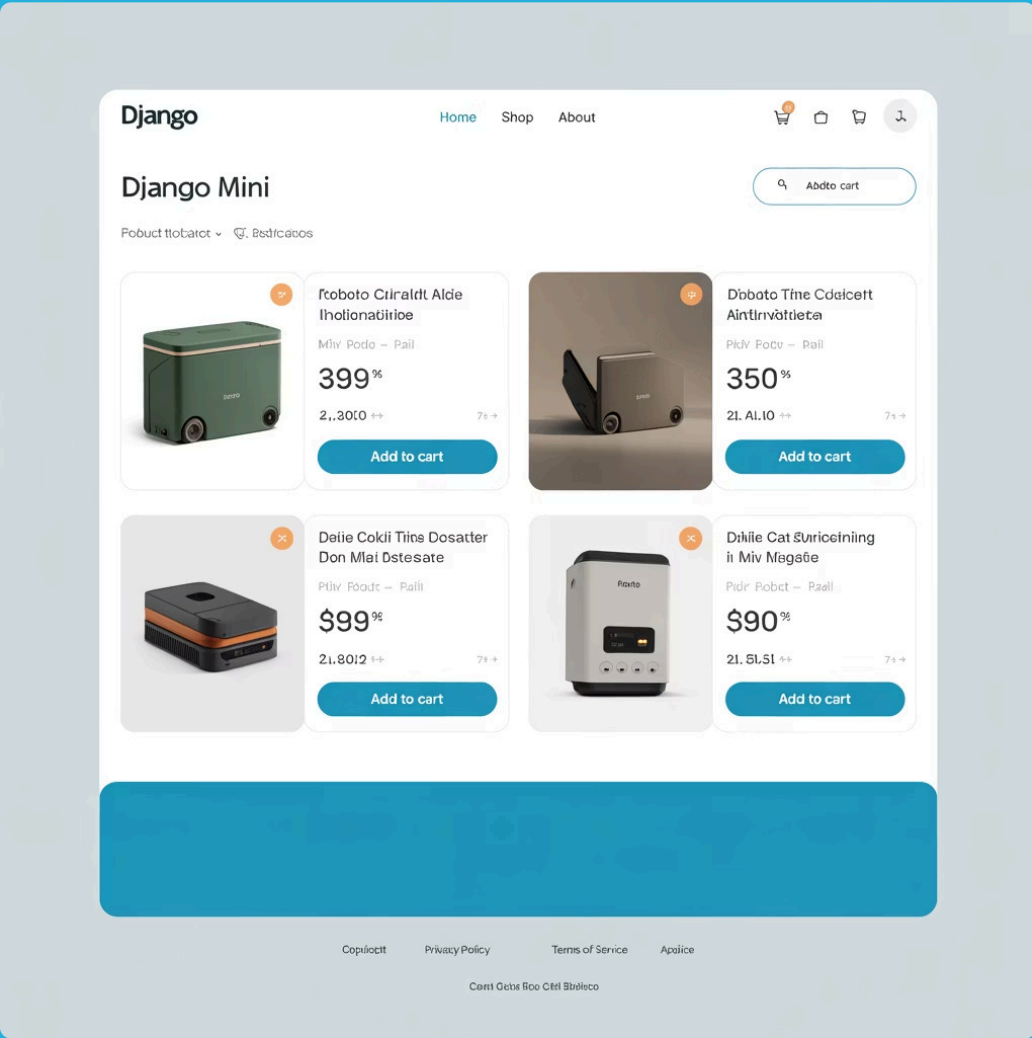
<div><div>Estructura Base</div><div>Crear base.html con navbar, contenido principal y footer reutilizable</div><div><pre><!DOCTYPE html> <html> <head> <title>{% block title %}Mi Tienda{% endblock %}</title> </head> <body> {% include 'navbar.html' %} <main> {% block content %}{% endblock %} </main> {% include 'footer.html' %} </body> </html></pre></div></div>	<div><div>Lista de Productos</div><div>Implementar iteradores y filtros para mostrar productos dinámicamente</div><div><pre>{% for producto in productos %} <div class="producto"> <h3>{{ producto.nombre capfirst }}</h3> <p>{{ producto.descripcion truncatewords :10 }}</p> \${{ producto.precio floatformat:2 }} </div> {% endfor %}</pre></div></div>	<div><div>Sistema de Login</div><div>Usar condiciones {% if %} para mostrar contenido según autenticación</div><div><pre>{% if user.is_authenticated %} <p>Bienvenido, {{ user.username }} </p> Cerrar Sesión {% else %} Iniciar Sesión {% endif %}</pre></div></div>
--	--	--

Componentes Requeridos


- base.html con bloques flexibles
- navbar.html con navegación
- footer.html con información
- productos.html con lista dinámica
- login.html con formulario

Funcionalidades

- Mostrar productos con filtros
- Navegación condicional
- Numeración automática
- Manejo de errores básico

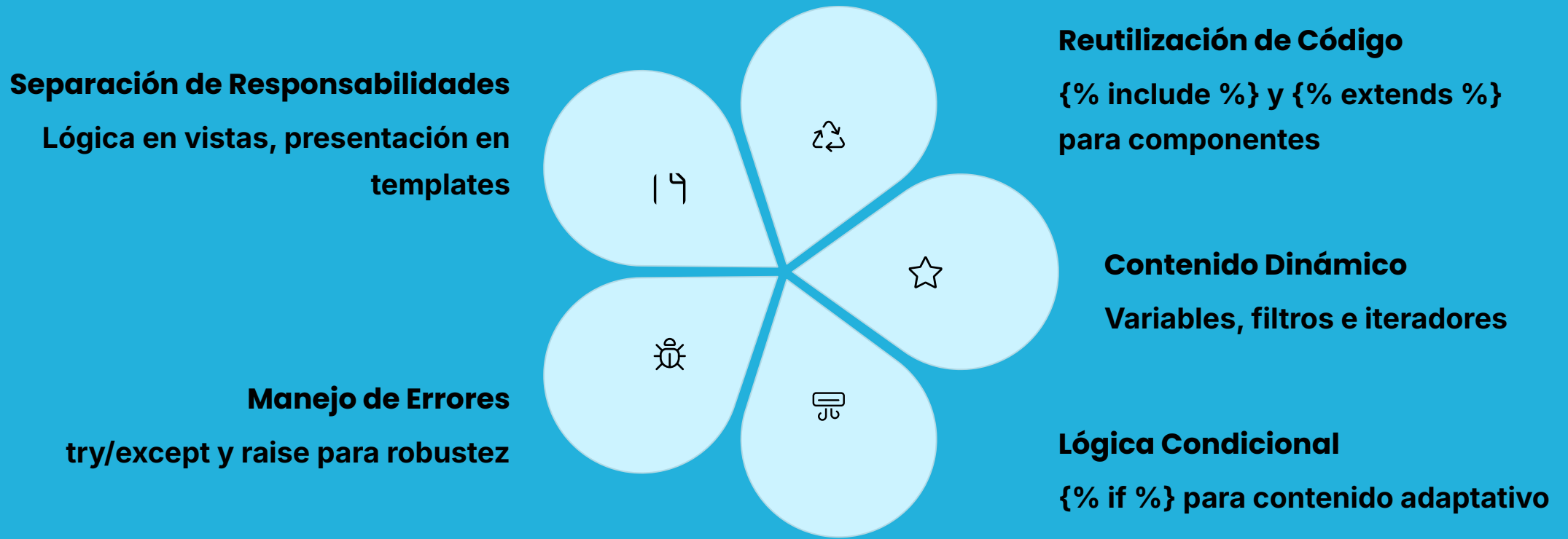


Entrega: Aplicación web funcional con renderización dinámica completa, navegación intuitiva y manejo de estados de usuario.

 Tiempo estimado: 45 minutos para implementación completa. Trabajar en equipos de 2-3 personas para maximizar el aprendizaje colaborativo.

Cierre y Conclusiones: Dominando Django Templates

Django Templates proporcionan un sistema poderoso y flexible para crear aplicaciones web dinámicas, separando efectivamente la lógica de la presentación.



Conceptos Clave Dominados

- **Variables:** {{ variable }} para datos dinámicos
- **Filtros:** |upper, |lower, |capfirst para transformaciones
- **Iteradores:** {% for %} con forloop.counter
- **Bloques:** {% block %} para herencia flexible
- **Condiciones:** {% if %} para lógica adaptativa
- **URLs:** {% url %} para enlaces dinámicos

Herramientas de Desarrollo



Teoría

Conceptos fundamentales



Práctica

Ejercicios activos



Templates

Plantillas dinámicas



VS Code

Entorno de desarrollo

¡Felicitaciones! Ahora tienes las herramientas necesarias para crear aplicaciones web dinámicas y profesionales con Django Templates. La separación de lógica y presentación te permitirá desarrollar proyectos más mantenibles y escalables.