POO · CEPY · UT4

Plantillas Jinja2

Separar la presentación HTML de la lógica del route

Programación de aplicaciones en Python — Sesión 4

Índice

1. **El problema:** HTML como string en el route
2. **Jinja2 +** `render_template`
3. **Sintaxis y herencia de plantillas**
4. **Adaptación de datos y malas prácticas**

> Continuación de la sesión 3. Asume que ya conoces routes, `url_for`, manejadores de error y logging.

1. El problema

HTML como string dentro del route

Así estábamos hasta ahora

```python
@app.route('/productos')
def listar_productos():
    html = "<h2>Lista de productos</h2><ul>"
    for t in servicio.listar_productos():
        html += f"<li>{t[0]} — {t[1]} — {t[3]:.2f} EUR</li>"
    html += "</ul>"
    return html
```

⚠️ **Inconvenientes**

El editor no resalta el HTML, los datos van por índice (`t[0]` , `t[3]`), y si la página crece a 80 líneas el route se vuelve ilegible.

La idea: separar estructura y datos

🚫 Antes

- HTML generado desde Python.
- Datos por índice de tupla o lista.
- Diseñador no puede tocar.

✅ Después

- HTML en ficheros `.html`.
- Route obtiene datos.
- Plantilla los muestra.

> Misma idea que la arquitectura por capas: cada cosa en su sitio.

2. Jinja2 y `render_template`

El motor de plantillas dinámicas que viene con Flask:
HTML que cambia según los datos del route

HTML estático vs HTML dinámico

HTML estático

- Fichero `.html` fijo en disco.
- Cambia solo al editarlo a mano.
- Igual para todos los usuarios.
- Lo serviría cualquier servidor web (Apache, nginx) sin Python.

HTML dinámico (Jinja)

- Plantilla `.html` **con huecos** rellenados en cada petición.
- Cambia con los datos del route (BD, estado, hora, usuario...).
- Distinta cada vez (precio actual, lista del momento...).
- Necesita un motor de plantillas + un framework como Flask.

Una web puramente estática se serviría como un fichero plano. **Jinja2 sirve para la parte dinámica**: cuando el HTML que se envía al navegador depende de datos que solo el servidor conoce en el momento de la petición.

El cambio en una línea

Route antiguo

```
@app.route('/saldo')
def saldo():
    s = servicio.saldo()
    return f"<p>{s:.2f} EUR</p>"
```

Route con plantilla

```
@app.route('/saldo')
def saldo():
    return render_template(
        'saldo.html',
        saldo=servicio.saldo()
    )
```

`render_template` busca el fichero en `templates/`, sustituye las marcas Jinja2 y devuelve el HTML.

Dónde se encuentran las plantillas

Flask espera una carpeta llamada **exactamente** `templates/` colgando del paquete:

```
expendedora/
└── presentation/
    ├── app.py
    └── templates/
        ├── base.html
        ├── productos.html
        └── saldo.html
```

⚠️ `TemplateNotFound`

Casi siempre: la carpeta no se llama `templates/` (plural, minúsculas) o no cuelga de `presentation/`.

3. Sintaxis Jinja2

`{{ }}`, `{% %}` y filtros con `|`

Las marcas de Jinja2

Marca	Para qué
`{{ variable }}`	**Imprime** un valor
`{% for %}` / `{% if %}`	**Estructuras de control**
`{% block %}`	**Hueco con nombre** para herencia
`{{ valor | filtro }}`	**Transforma** antes de imprimir

> Cada bloque (`for`, `if`, `block`) se cierra con su `{% end... %}`. **Importante: `{% for %}`, `{% if %}` y los filtros son sintaxis de Jinja2, no de Python** — se parecen porque Jinja se inspira en Python, pero el motor que las interpreta es Jinja2, no el intérprete Python.

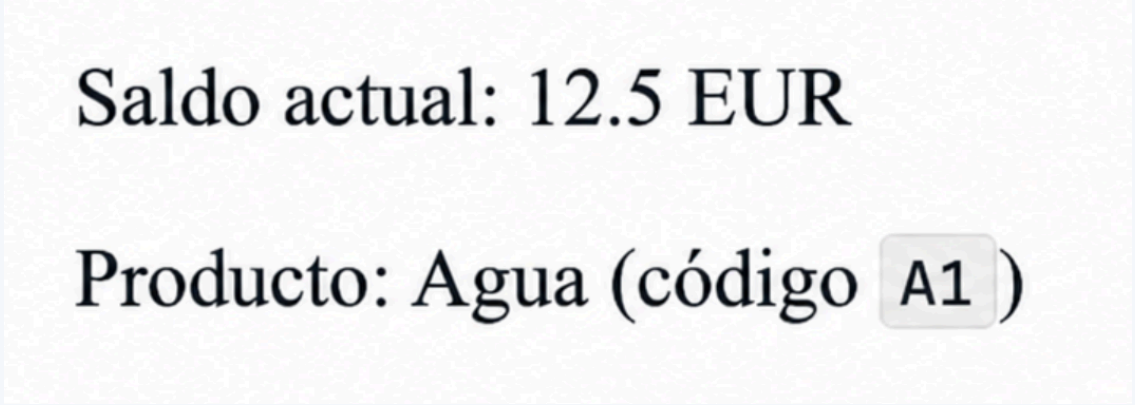
`{{ }}`: qué se muestra en el navegador

Plantilla

```
<p>Saldo actual: {{ saldo }} EUR</p>
<p>Producto: {{ producto.nombre }}
   (código <code>{{ producto.codigo }}</code>)</p>
```

```
render_template(
    'producto.html',
    saldo=12.5,
    producto={'codigo': 'A1',
              'nombre': 'Agua'},
)
```

Resultado



Cada `{{ … }}` se sustituye por el valor de la variable. El navegador no recibe ningún `{{ }}`, solo el HTML final.

Estructura de control `{% if else endif %}` simple

```
{% if saldo %}
<p>Tienes {{ saldo }} EUR insertados.</p>
{% else %}
<p>Inserta dinero para comprar.</p>
{% endif %}
```

✅ Con `saldo=2.5`

`<p>Tienes 2.5 EUR insertados.</p>`

⚠️ Con `saldo=0`

`<p>Inserta dinero para comprar.</p>`

Una lista vacía, `0`, `None` o `""` son **falsos** en Jinja2 igual que en Python.

Estructura de control `{% for endfor %}` simple

Plantilla

```
<ul>
{% for p in productos %}
    <li>{{ p.codigo }} -
        {{ p.nombre }}</li>
{% endfor %}
</ul>
```

HTML resultante

```
<ul>
    <li>A1 - Agua</li>
    <li>B1 - Refresco</li>
    <li>C1 - Chocolatina</li>
</ul>
```

Recorre una lista igual que en Python. **Siempre se cierra con `{% endfor %}`** : las plantillas no usan la indentación de Python para delimitar bloques, sino marcas explícitas (`{% endif %}` , `{% endfor %}` , `{% endblock %}`) porque viven dentro de HTML, que ignora la indentación.

`{% for endfor %}` en el navegador

- A1 — Agua (12 uds.)
- B2 — Refresco (8 uds.)
- C3 — Chocolatina (20 uds.)

`{% if else endif %}` + `{% for endfor %}` anidados

Plantilla

```
{% if productos %}
<ul>
    {% for p in productos %}
        <li>{{ p.codigo }} –
            {{ p.nombre }}</li>
    {% endfor %}
</ul>
{% else %}
<p>No hay productos.</p>
{% endif %}
```

Con productos

```
<ul>
    <li>A1 – Agua</li>
    <li>B1 – Refresco</li>
</ul>
```

Con lista vacía

```
<p>No hay productos.</p>
```

Las estructuras se anidan como en Python. `{% if productos %}` cubre el caso "lista vacía" sin escribir `len(productos) > 0`.

Filtros: el operador `|`

Un **filtro** transforma un valor antes de imprimirlo. Se aplica con `|` y se lee **de izquierda a derecha**:

```
<p>{{ nombre|upper }}</p>
```

Si `nombre = "Agua"`, Muestra: `<p>AGUA</p>`.

> 💡 **Los filtros se pueden encadenar**
> Conectando varios con `|`, la salida de uno pasa al siguiente como una tubería: `valor | filtro1 | filtro2 | filtro3`. Por ejemplo, `{{ "Agua"|upper|replace("A", "*") }}` produce `*GU*`.

Filtros más usados de Jinja2

Filtro	Para qué	Ejemplo (entrada → salida)
`|upper` / `|lower`	Mayúsculas / minúsculas	`"Agua"|upper` → `"AGUA"`
`|length`	Longitud de lista/cadena	`[1,2,3]|length` → `3`
`|default(x)`	Valor por defecto si está vacío	`None|default("–")` → `"–"`
`|format( ... )`	Formato estilo `printf`	`"%.2f"|format(1.5)` → `"1.50"`
`|replace(a, b)`	Sustituir	`"1.50"|replace(".", ",")` → `"1,50"`
`|join(", ")`	Unir lista con separador	`["a","b"]|join(", ")` → `"a, b"`

> Hay muchos más en la documentación de Jinja2.

Encadenar filtros

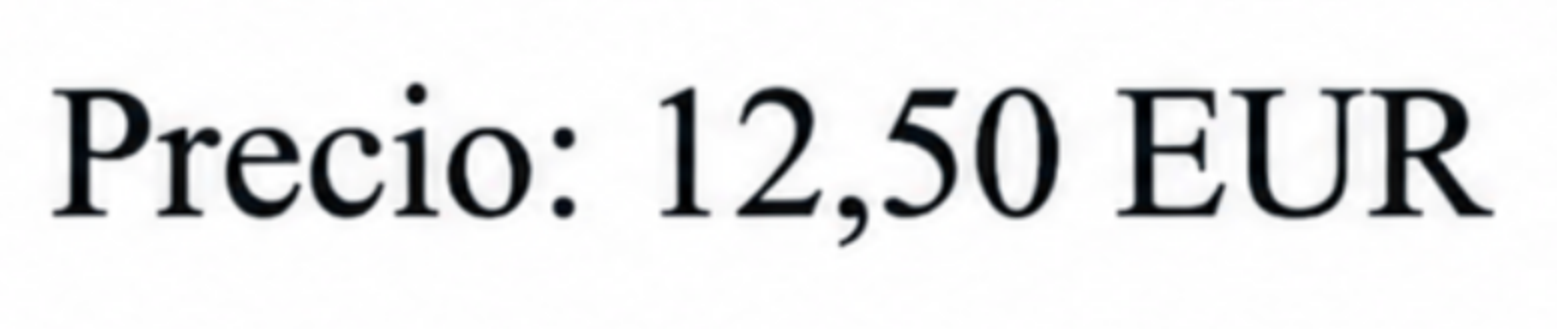
Los filtros se pueden conectar con `|`: la salida de uno entra como entrada del siguiente, de izquierda a derecha. Caso típico — un precio con dos decimales y coma:

```
<p>Precio: {{ "%.2f"|format(precio)|replace(".", ",") }} EUR</p>
```

Con `precio = 1.5`, el flujo es:

Paso	Valor en juego
Entrada	`1.5`
Tras `"%.2f"|format( ... )`	`"1.50"`
Tras `|replace(".", ",")`	`"1,50"`
HTML final	`<p>Precio: 1,50 EUR</p>`

Encadenar filtros. Ejemplo



> El navegador recibe `<p>Precio: 1,50 EUR</p>`: el resultado de encadenar los dos filtros sobre `1.5`.

4. Herencia de plantillas

Compartir cabecera y navegación sin duplicar código

El problema: cabecera repetida en cada página

Cada página de la web necesita la **misma estructura**: `<!doctype>`, `<head>` con `<title>`, cabecera con logo y navegación, pie. Si copiamos ese esqueleto en cada plantilla...

- Una sola lista de productos: cabecera + tabla.
- Un detalle de producto: cabecera + ficha.
- Una página de saldo: cabecera + número.
- Una página de ayuda: cabecera + listado de rutas.
- Una página de error 404: cabecera + mensaje.

Cualquier cambio (un menú nuevo, un logo, una hoja de estilos) obliga a tocar **todas** las plantillas. Misma trampa que copiar código entre clases en POO.

✅ **Solución: herencia de plantillas**
Una plantilla **padre** (`base.html`) define la estructura común. Cada plantilla **hija** solo escribe lo que cambia — el resto lo hereda. Cambiar la cabecera = tocar un solo fichero.

Padre e hija lado a lado

`base.html` (padre)

```
<!doctype html>
<html lang="es">
<head>
  <title>{% block titulo %}
    Expendedora
  {% endblock %}</title>
</head>
<body>
  <header> … </header>
  <main>
    {% block contenido %}
    {% endblock %}
  </main>
</body>
</html>
```

`saldo.html` (hija)

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

{% block titulo %}
  Saldo – Expendedora
{% endblock %}

{% block contenido %}
  <h2>Saldo actual</h2>
  <p>
    {{ "%.2f"|format(saldo)|replace(".", ",") }} EUR
  </p>
{% endblock %}
```

Herencia: la fusión final

HTML enviado al navegador

```
<!doctype html>
<html lang="es">
<head>
    <title>Saldo – Expendedora</title>
</head>
<body>
    <header>
        <h1><a href="/">Expendedora</a></h1>
        <nav>
            <a href="/productos">Productos</a> ·
            <a href="/saldo">Saldo</a> ·
            <a href="/ayuda">Ayuda</a>
        </nav>
    </header>
    <main>
        <h2>Saldo actual</h2>
        <p><strong>12,50 EUR</strong></p>
    </main>
</body>
```

En el navegador

Contenido fuera de `{% block %}` se ignora

Plantilla hija

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

<p>NO aparece (fuera de block).</p>

{% block contenido %}
<p>SÍ aparece (dentro de block).</p>
{% endblock %}
```

HTML resultante

```
...
<main>
  <p>SÍ aparece (dentro de block).</p>
</main>
...
```

⚠️ **A revisar al modificar plantillas hijas**

Jinja2 no avisa: ignora silenciosamente el contenido fuera de `{% block %}`. Si "falta" algo en la página, comprueba primero si lo has puesto fuera de un bloque.

Relacioanando con herencia en POO

💡 **La misma idea**

- `base.html` → la **clase padre** con la estructura común.
- Cada `{% block %}` → un **método sobrescribible**.
- La hija → **redefine** solo lo que cambia, hereda el resto.

> Si una página pierde la cabecera, casi siempre falta `{% extends "base.html" %}` en la primera línea.

5. Adaptar tuplas para la plantilla

`tupla → dict` en el route

El servicio devuelve tuplas

```
(codigo, nombre, precio_base, precio_final, cantidad, descuento)
```

🚫 Ilegible en la plantilla

```
<td>{{ p[0] }}</td>
<td>{{ p[1] }}</td>
<td>{{ p[3] }}</td>
```

Hay que contar campos uno a uno.

✅ Legible

```
<td>{{ p.codigo }}</td>
<td>{{ p.nombre }}</td>
<td>{{ p.precio_final }}</td>
```

La conversión se hace en el route

```python
@app.route('/productos')
def listar_productos():
    keys = ['codigo', 'nombre', 'precio_base',
            'precio_final', 'cantidad', 'descuento']
    productos = [dict(zip(keys, t))
                 for t in servicio.listar_productos()]
    return render_template('productos.html', productos=productos)
```

💡 Qué hace `dict(zip(keys, t))`

```
>>> dict(zip(['codigo', 'nombre'], ('A1', 'Agua')))
{'codigo': 'A1', 'nombre': 'Agua'}
```

`zip` empareja, `dict` recoge. La comprensión repite para cada tupla.

La conversión es **adaptación a la presentación**, por eso se hace en el route, no en el servicio.

`url_for` en plantillas

Igual que rutas para construir enlaces, pero ahora dentro del HTML:

```
<a href="{{ url_for('inicio') }}">Inicio</a>
<a href="{{ url_for('ver_producto', codigo=p.codigo) }}">
  Detalle
</a>
```

💡 **Por qué no escribir la URL directamente**

Recuerda que lo hacíamos así porque si renombras el patrón de la ruta, los enlaces hechos con `url_for` se actualizan solos. Los escritos a mano se rompen sin avisar.

6. Malas prácticas

Lógica de negocio en la plantilla

La plantilla muestra, no decide

🚫 Incorrecto

```
<td>
  {{ "%.2f"|format(
    p.precio_base *
    (1 - p.descuento / 100)
  ) }} EUR
</td>
```

La plantilla aplica el descuento.

✅ Correcto

```
<td>
  {{ "%.2f"|format(p.precio_final)|replace(".", ",") }} EUR
</td>
```

El **dominio** ya entrega el precio final.

Si la lógica está en la plantilla, no se testea, no se reutiliza desde otra interfaz (CLI, API JSON) y se mezcla con el HTML.

Verificación rápida

⚠️ **Si en una plantilla ves `servicio.algo()`...**

...la lógica está en el sitio equivocado. La plantilla recibe datos ya preparados, no los va a buscar ella.

Las únicas funciones que se llaman desde la plantilla son las **utilidades de Jinja2/Flask**:

- `url_for`
- `|format`
- `|replace`
- `|length`
- `|upper`

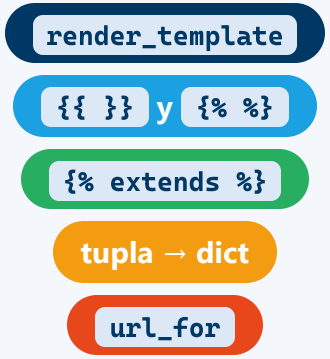
Resumen

- HTML fuera del route → fichero `.html` con huecos.
- `render_template('plantilla.html', var=valor)` rellena y devuelve.
- `{{ }}` imprime, `{% %}` controla, `|filtro` transforma.
- `{% extends %}` + `{% block %}` evita duplicar la cabecera.
- **Tupla → dict** en el route para que la plantilla sea legible.
- La plantilla **muestra**, el route **adapta**, el servicio **decide**.

✅ **Alcance del lab 4**

Aplicaremos estas plantillas a las páginas de **lectura** de la expendedora: `base.html`, `productos.html`, `producto.html`, `saldo.html`, `buscar.html`, `inicio.html`, `ayuda.html` y `error.html`. Las páginas de **escritura** (insertar, comprar, agregar...) llegarán en lab 5 con los formularios HTML.

¿Preguntas?



POO · CEPY · UT4 — Plantillas Jinja2