POO · CEPY · UT4

Formularios HTML y método POST

Datos del usuario en Flask

Programación de aplicaciones en Python — Sesión 5

Índice

> Continuación de la sesión 4 (plantillas Jinja2). Asumimos que ya conocemos routes, `try/except` de dominio y `render_template`.

1. GET vs POST

Por qué los datos de escritura no caben en la URL

Lo que veníamos haciendo

Hasta ahora, cuando una acción modificaba el estado del servidor, pasábamos los datos **en la propia URL**:

```
GET /insertar/2.0
```

⚠️ **Funciona, pero…**

- La URL queda **visible** en la barra y en el historial.
- Pulsa **F5** y vuelves a insertar 2 EUR. Sin avisar.
- Cualquier precarga del navegador, indexador o vista previa de chat dispara la acción.
- Tamaño limitado y restricciones de caracteres.

> `GET` vale para **leer**. Para **escribir** necesitamos otra cosa.

GET vs POST de un vistazo

`GET`

- Datos en la **URL**.
- Visible y cacheable.
- F5 repite **sin preguntar**.
- Idempotente: leer no cambia nada.
- Para **leer**: mostrar, consultar, listar.

`POST`

- Datos en el **cuerpo** de la petición.
- No visibles en la barra.
- F5 hace al navegador **preguntar**.
- No idempotente: escribir cambia estado.
- Para **escribir**: crear, modificar, borrar.

Regla práctica

Si la URL...	Verbo
Solo lee algo y volver a llamarla no cambia nada	`GET`
Escribe en la base de datos: alta, modificación, borrado	`GET` + `POST`

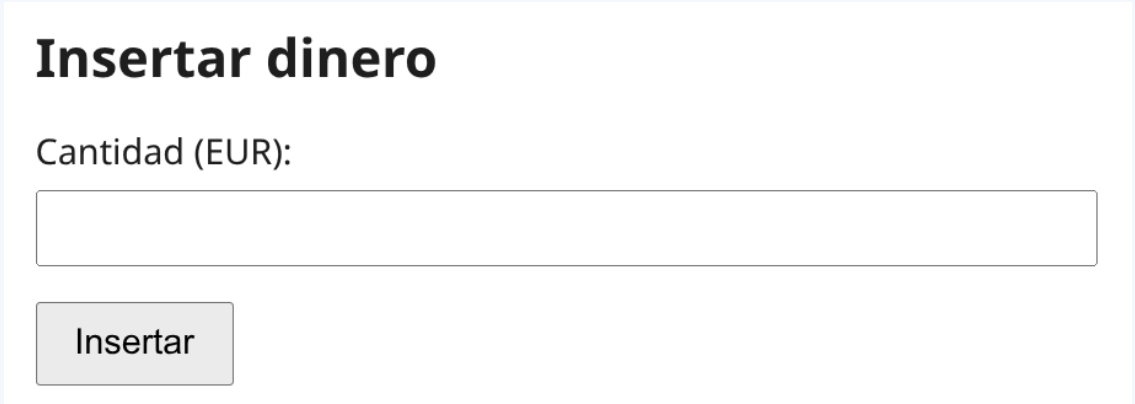
> Las rutas que muestran datos (listados, fichas, búsquedas) se quedan en GET.
> Las que tocan estado genera vista para leer datos con GET se procesan con POST.

2. El formulario HTML

`<form>`, `<input>`, `request.form`

Anatomía mínima de un formulario

```
<form method="post"
      action="{{ url_for('insertar') }}">
    <label>
        Cantidad (EUR):
        <input type="text"
               name="cantidad" required>
    </label>
    <button type="submit">Insertar</button>
</form>
```



Sin `name`, el campo **no se envía**. Sin `name`, no existe.

Atributos clave del formulario

Atributo	Qué hace
`method="post"`	Verbo HTTP del envío.
`action="..."`	URL destino (siempre con `url_for`).
`name="cantidad"`	**Clave** con la que el campo llega al servidor.
`required`	Validación en cliente: el navegador no deja enviar vacío.

Cliente vs servidor: dónde se valida

Cliente (navegador)

- `required`, `type="number"`, `min/max`, `pattern`…
- Rechaza el envío **antes de que salga** del equipo.
- Rápida, mejora la experiencia.
- **Se puede saltar**: DevTools, JS off, `curl`.

Servidor (Flask + dominio)

- `try/except` con `ValueError` y excepciones de dominio.
- Se ejecuta **sí o sí** al llegar la petición.
- La que protege los datos de verdad.

> **Cliente = ayuda. Servidor = defensa.** Lo que pongamos en el HTML no nos exime de comprobarlo en Python.

`request.form`: el diccionario del POST

Cuando llega un POST, Flask construye un **diccionario** `request.form` con los campos:

```
from flask import request

codigo   = request.form['codigo']            # KeyError si no llega
cantidad = request.form.get('cantidad', '')  # '' si no llega
```

Acceso	Comportamiento
`request.form['x']`	Lanza `KeyError` si la clave no existe.
`request.form.get('x', '')`	Devuelve el valor por defecto si no existe.

Ya conocíamos `request` de sesiones anteriores (`request.method` , `request.path`). Ahora usamos propiedad más: `request.form` .

Una ruta, dos verbos

El **mismo route** muestra el formulario (GET) y lo procesa (POST):

```python
@app.route('/insertar', methods=['GET', 'POST'])
def insertar():
    if request.method == 'POST':
        # procesar el formulario y redirigir
        ...
    # GET: mostrar el formulario
    return render_template('insertar.html', error=None, cantidad=None)
```

💡 **`methods=['GET', 'POST']`**

Lista los verbos permitidos sobre esa URL. Cualquier otro verbo (p.ej. `DELETE`) recibe `405 Method Not Allowed` automáticamente.

3. Patrón Post/Redirect/Get

Tras un POST con éxito, **no devuelvas HTML**

El problema sin PRG

```python
# Mala práctica
@app.route('/insertar', methods=['GET', 'POST'])
def insertar():
    if request.method == 'POST':
        cantidad = float(request.form['cantidad'])
        servicio.insertar_dinero(cantidad)
        return render_template('saldo.html', saldo=servicio.saldo())  # mal
    return render_template('insertar.html')
```

🚫 Qué ve el usuario

- La URL sigue siendo `/insertar`.
- El navegador recuerda que la última petición fue un POST con `cantidad=2.0`.
- Pulsa **F5** → *"¿Reenviar formulario?"* → si dice que sí, **inserta otros 2 EUR**.

La solución: redirigir tras éxito

```python
@app.route('/insertar', methods=['GET', 'POST'])
def insertar():
    if request.method == 'POST':
        cantidad = float(request.form['cantidad'])
        servicio.insertar_dinero(cantidad)
        return redirect(url_for('saldo'))   # ⟵ la clave
    return render_template('insertar.html')
```

✅ **Qué pasa entonces**

1. El navegador recibe `302 Found` con `Location: /saldo`.
2. Hace un **GET** a `/saldo`.
3. Esa GET es la que aparece en la barra de direcciones.
4. F5 recarga `/saldo` (un GET inocuo), no el POST original.

Tras un POST con éxito, no devuelvas HTML: devuelve una redirección.

La tabla del PRG

Situación	Qué devolver
GET (mostrar formulario)	`render_template('form.html', ...)`
POST con **éxito**	`redirect(url_for('destino'))`
POST con **error**	`render_template('form.html', error=..., datos=...)`, código 400/404/409

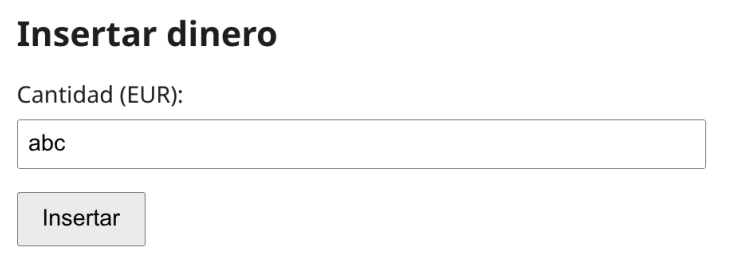
> En **éxito** cambia de URL. En **error** se queda en la misma con los datos rellenos.

Validación con re-render

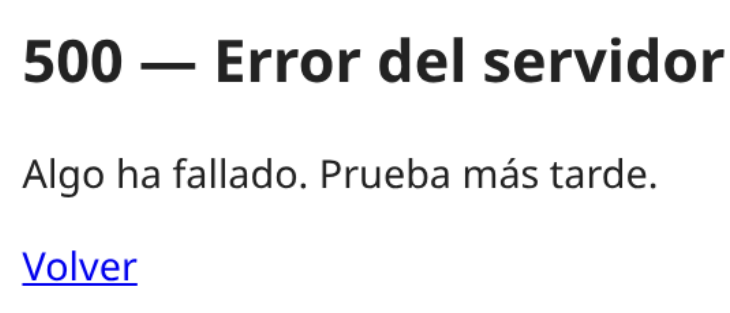
Mismo `try/except` de siempre, distinto `except`

El usuario teclea algo inválido

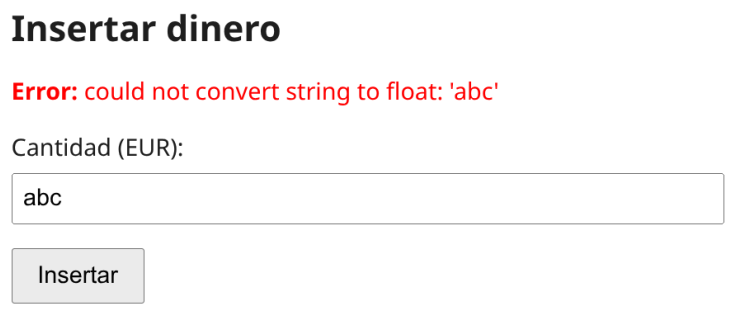
1. Valor incorrecto



2. Sin try/except: 500



3. Con re-render



> Sin captura: error genérico y pierde lo tecleado. Con re-render: mensaje claro y el valor sigue ahí.

Validar es lo que ya sabemos

La novedad **no es** dónde se valida — sigue en el dominio, como siempre.
La novedad es **qué se devuelve** cuando falla:

Tipo	Origen	Excepción
Formato	`float('abc')`, `int('1.5')`	`ValueError`
Negocio	precio ≤ 0, cantidad < 0	`ValueError`
Estado	código repetido	`ProductoYaExisteError`
Estado	código que no existe	`ProductoNoEncontradoError`

> Las dos primeras lanzan **el mismo tipo** por casualidad útil: al usuario le da igual si tecleó `abc` o `-1`. Un único `except ValueError` cubre las dos.

Plantilla (1/2): mostrar el error

Encima del `<form>`, un bloque `{% if error %}` que solo se dibuja cuando llega:

```
{% if error %}
<p style="color: red"><strong>Error:</strong> {{ error }}</p>
{% endif %}

<form method="post" action="{{ url_for('insertar') }}">
    ...
</form>
```

> Cuando el route pasa `error=None` (GET o éxito), Jinja2 evalúa el `if` como falso y el `<p>` no se renderiza.

Plantilla (2/2): conservar lo tecleado

Dentro del `<input>`, un atributo `value` que se rellena con el dato previo:

```
<input type="text" name="cantidad" value="{{ cantidad or '' }}" required>
```

💡 **`{{ cantidad or '' }}`**

- Si `cantidad` viene con valor → se usa ese valor.
- Si `cantidad` viene a `None` → Python evalúa el `or` y devuelve `''`.
- Resultado: `value=""` y el campo aparece limpio en el GET inicial.

Patrón completo: `/insertar`

```python
@app.route('/insertar', methods=['GET', 'POST'])
def insertar():
    if request.method == 'POST':
        try:
            cantidad = float(request.form['cantidad'])
            servicio.insertar_dinero(cantidad)
            return redirect(url_for('saldo'))                # éxito: PRG
        except ValueError as e:
            return render_template(
                'insertar.html',
                error=str(e),
                cantidad=request.form.get('cantidad', '')   # conserva lo tecleado
            ), 400
    return render_template('insertar.html', error=None, cantidad=None)  # GET
```

En **éxito** redirigimos. En **error** re-renderizamos con `error=` y `cantidad=`.

Escenarios típicos en routes de escritura

Escenario	Excepción	Código HTTP	Render
`cantidad=abc`	`ValueError` (formato)	`400`	Formulario + error
`cantidad=-1`	`ValueError` (negocio)	`400`	Formulario + error
Código repetido	`ProductoYaExisteError`	`409`	Formulario + error
Código inexistente	`ProductoNoEncontradoError`	`404`	Formulario + error

`ValueError` aparece en cualquier route con conversión o validación numérica (como `/insertar`). `ProductoYaExisteError` y `ProductoNoEncontradoError` viven en routes que crean o borran recursos (`/agregar`, `/seleccionar`, `/eliminar`).

Un route, varios `except` apilados

Cuando un route puede fallar de varias formas, **un `try` con varios `except`** — cada uno con su código HTTP:

```python
@app.route('/agregar', methods=['GET', 'POST'])
def agregar():
    if request.method == 'POST':
        datos = request.form
        try:
            precio = float(datos['precio'])
            servicio.agregar_producto(datos['codigo'], datos['nombre'], precio, ...)
            return redirect(url_for('listar_productos'))
        except ValueError as e:                          # formato o negocio
            return render_template('agregar.html', datos=datos, error=str(e)), 400
        except ProductoYaExisteError as e:               # conflicto de estado
            return render_template('agregar.html', datos=datos, error=str(e)), 409
    return render_template('agregar.html', datos={}, error=None)
```

Mismo patrón siempre: re-render con `error=` + datos previos, pero cambia el **código HTTP** según el tipo de fallo.

Conservar siempre lo tecleado

⚠️ **El error acompaña los datos, no los sustituye**

Si el usuario escribió `1,5` (con coma) y le dices *"formato incorrecto"*, lo último que quiere es **escribirlo todo otra vez**.

En formularios con varios campos (`/agregar`), pasamos `datos=request.form` entero a la plantilla y la plantilla lee `{{ datos.codigo or '' }}`, `{{ datos.nombre or '' }}`, etc.

4. Operaciones destructivas y POST

Borrar **nunca** puede ir en GET

Por qué un enlace que borra es peligroso

Un `<a href="/eliminar/A1">` se puede activar **sin intención del usuario**:

- **Clic accidental** en el enlace.
- **F5** tras una eliminación previa.
- **Vista previa** del enlace en chat (Slack, Telegram hacen un GET silencioso).
- **Indexación** por buscadores en intranet.
- **Precarga** del navegador.

> Todos esos casos lanzan un `GET` automático. Si el GET borra, el daño está hecho.

Mala práctica vs práctica correcta

🚫 Mal: borrar con `<a>`

```
<a href="/eliminar/A1">
  Eliminar
</a>
```

Cualquier GET automático ejecuta el borrado.

✅ Bien: borrar con `<form>` POST

```
<form method="post"
  action="{{ url_for('eliminar',
                     codigo='A1') }}">
  <button type="submit">
    Sí, eliminar
  </button>
</form>
```

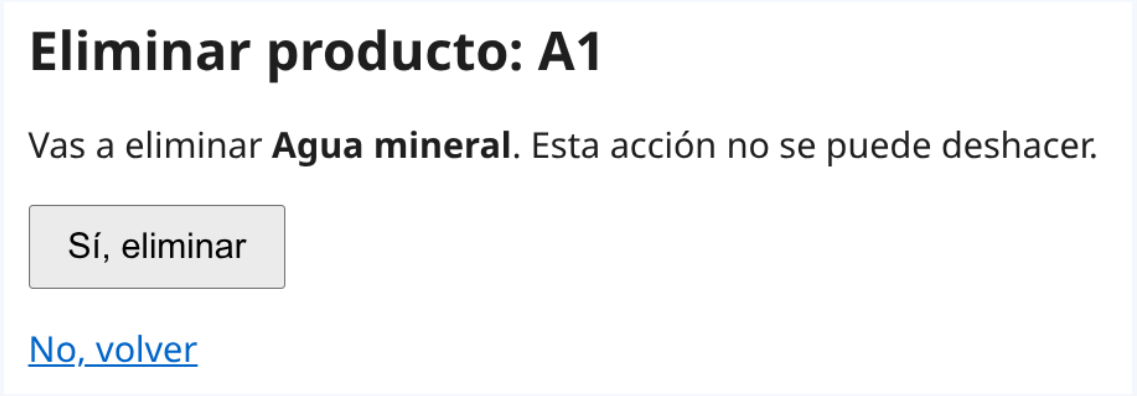
Solo se dispara al pulsar el botón.

Patrón GET-confirma + POST-actúa

Dos pasos sobre la **misma URL**:

1. **GET** `/eliminar/A1` → confirmación con el **qué** se va a borrar.
2. **POST** `/eliminar/A1` → ejecuta y redirige.

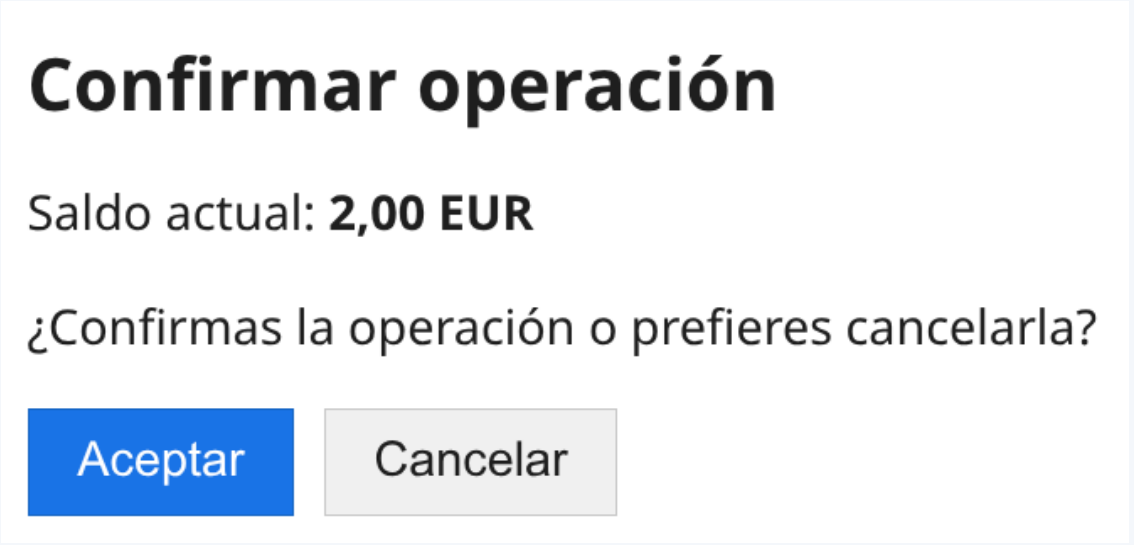
```
@app.route('/eliminar/<codigo>',
           methods=['GET', 'POST'])
def eliminar(codigo):
    if request.method == 'POST':
        servicio.eliminar_producto(codigo)
        return redirect(url_for(
            'listar_productos'))
    # GET: pantalla de confirmación
    producto = ...
    return render_template(
        'eliminar.html',
        producto=producto)
```



Routes que solo aceptan POST

`/cancelar` no necesita confirmación previa: el botón ya vive junto al de confirmar.

```
@app.route('/cancelar',
           methods=['POST'])
def cancelar():
    devuelto = servicio.cancelar()
    return render_template(
        'mensaje.html', ... )
```



💡 **Si alguien teclea `http://localhost:5000/cancelar` en la barra…**

Flask responde `405 Method Not Allowed`. La única forma de cancelar es pulsando el botón del formulario.

Resumen: el patrón completo

Pieza	Dónde vive	Qué hace
`<form method="post" action="...">`	Template	Empaqueta y envía los campos.
`name="..."` en cada `<input>`	Template	Define la clave del campo.
`methods=['GET', 'POST']`	Route	Permite los dos verbos sobre la misma URL.
`if request.method == 'POST':`	Route	Bifurca entre mostrar y procesar.
`request.form['x']` / `.get('x', '')`	Route	Lee los campos enviados.
`try/except` con dominio	Route	Captura formato, negocio y estado.
`return redirect(url_for(...))`	Route	PRG tras éxito.
`render_template(..., datos=..., error=...)`	Route	Re-render tras error.

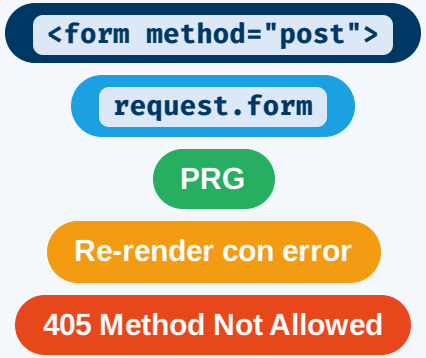
Qué hemos conseguido

🎯 Una app web honesta

- Ninguna acción de escritura viaja por la URL.
- F5 deja de duplicar operaciones (PRG).
- Los errores re-renderizan el formulario con lo tecleado y un mensaje claro.
- Las operaciones destructivas exigen confirmación y botón POST.

Quedan dos mejoras para más adelante: **mensajes flash** (avisos tras un redirect) y **traducir los mensajes de excepción** de Python en algo más amigable para el usuario.

¿Preguntas?


<form method="post">
request.form
PRG
Re-render con error
405 Method Not Allowed