POO · CEPY · UT4

Rutas dinámicas y excepciones

Flask: converters, códigos HTTP, estado y formularios

Programación de aplicaciones en Python — Sesión 2

Índice

1. Tipos en los parámetros de URL

Converters, regex internas y dónde termina Flask

El recordatorio: `<codigo>` es texto

```
@app.route("/producto/<codigo>")
def ver_producto(codigo):
    return f"Has pedido el producto {codigo}"
```

Lo que captura `<codigo>` es **siempre una cadena**:

- `/producto/A1` → `codigo = "A1"`
- `/producto/12` → `codigo = "12"` (string, **no** número)

> Suficiente para códigos y nombres. Insuficiente cuando esperas un número para operar con él.

Converters de Flask

Un **converter** valida el tipo del segmento **antes** de llamar a tu función. Si no encaja, Flask responde 404 automático.

```
@app.route("/comprar/<int:cantidad>")
def comprar(cantidad):
    return f"Compras {cantidad} unidades"  ## cantidad ya es int
```

Converter	Qué acepta	Uso típico
`<string:x>`	Texto sin `/` (por defecto)	Códigos, nombres
`<int:x>`	Enteros sin signo	Cantidades, IDs
`<float:x>`	Decimales con punto	Precios
`<path:x>`	Texto que incluye `/`	Rutas anidadas
`<uuid:x>`	Identificadores UUID	IDs únicos

Las regex detrás de los converters

Cada converter usa una **expresión regular** (patrón que describe qué forma tiene que tener el texto):

Converter	Regex	Qué significa
`<int:x>`	`\d+`	Solo dígitos. **No acepta signo ni decimales**
`<float:x>`	`\d+\.\d+`	Dígitos, punto, dígitos. **Exige decimal**
`<string:x>`	`[^/]+`	Cualquier cosa menos `/`

⚠️ **Consecuencia sorprendente**

`/comprar/-1` → 404 (el `-` no es dígito). `/insertar/1` → 404 (`<float>` exige el punto).

Ejemplos reales del comportamiento

URL	Resultado	Por qué
`/comprar/3`	✅ `cantidad=3`	Satisface `\d+`
`/comprar/tres`	❌ 404	No son dígitos
`/comprar/-1`	❌ 404	El signo `-` no es dígito
`/insertar/1.5`	✅ `cantidad=1.5`	Satisface `\d+\.\d+`
`/insertar/1`	❌ 404	Falta el punto decimal
`/insertar/-1.5`	❌ 404	El signo `-` no es dígito

> Si el converter rechaza, tu función **ni se ejecuta**. Los `try/except ValueError` dentro del route nunca se disparan para valores inválidos de tipo.

Flask filtra tipo, dominio filtra negocio

```python
@app.route("/insertar/<float:cantidad>")
def insertar(cantidad):
    try:
        servicio.insertar_dinero(cantidad)   ## ValueError si es ⩽ 0
        return f"Saldo: {servicio.saldo():.2f} EUR"
    except ValueError as e:
        return str(e), 400
```

- `/insertar/0.0` → llega al dominio, lanza `ValueError` → **400**
- `/insertar/-1.5` → Flask lo rechaza → **404** antes de llegar

🚫 **Mala práctica:** poner la validación de tipo en el dominio.

Deja que Flask filtre el formato; el dominio se ocupa solo del significado.

2. Códigos HTTP como lenguaje común

Cada respuesta empieza con un número que resume qué ha pasado

Anatomía de una respuesta HTTP

```
HTTP/1.1 200 OK                ← código de estado + descripción
Content-Type: text/html

<h1>Productos</h1>             ← cuerpo
```

El navegador **lee el código antes que el cuerpo**. Los códigos se agrupan por la primera cifra:

Rango	Familia	Significado
`2xx`	Éxito	Todo correcto
`3xx`	Redirección	Ve a otra URL
`4xx`	Error del cliente	La petición viene mal
`5xx`	Error del servidor	El servidor ha fallado

Los códigos que usarás en el proyecto

Código	Nombre	Cuándo lo devuelves
`200`	OK	Respuesta normal con contenido
`302`	Found	Tras una acción, redirección
`400`	Bad Request	Datos inválidos del cliente
`404`	Not Found	Recurso no existe
`409`	Conflict	Choca con el estado actual
`500`	Server Error	Excepción no capturada

> La elección **no es arbitraria**. Navegadores y motores de búsqueda actúan según el código recibido.

Devolver un código desde un route

Por defecto Flask añade `200 OK`. Para devolver otro código, `return` acepta una tupla `(cuerpo, codigo)`:

```
@app.route("/agregar/<codigo>/ ... ")
def agregar(codigo, ... ):
    try:
        servicio.agregar_producto(codigo, ... )
        return redirect(url_for("ver_producto", codigo=codigo))
    except ProductoYaExisteError as e:
        return str(e), 409
```

El código va como **segundo elemento** de la tupla. Flask añade la descripción textual (`"Conflict"` , `"Not Found"` ...) a partir del número.

Elegir entre 400, 404 y 409

Situación	Código	Por qué
Producto inexistente	`404`	El recurso pedido no existe
Precio de alta = 0	`400`	Los datos son inválidos
Alta con código ya existente	`409`	Conflicto con el estado del servidor
Excepción no capturada	`500`	Fallo interno no previsto

✅ **Heurística**

`400` → datos malos. `404` → recurso ausente. `409` → el servidor ya tiene algo que choca.

3. Redirección y URLs declarativas

`redirect`, `url_for` y el patrón "actúa → redirige"

El código 302 y `redirect`

Un **redirect** es una respuesta HTTP que dice al navegador: "no estoy aquí, ve a esta otra URL". Código habitual: `302 Found`.

```
from flask import redirect

@app.route("/comprar")
def comprar():
    servicio.comprar()
    return redirect("/productos")   ## el navegador carga /productos
```

🚫 **Problema: URL literal.**
Si renombras `/productos` a `/catalogo`, rompes todos los `redirect("/productos")` del código.

`url_for` — URLs por nombre de función

`url_for()` construye la URL a partir **del nombre de la función**, no del string.

```
from flask import redirect, url_for

@app.route("/comprar")
def comprar():
    servicio.comprar()
    return redirect(url_for("listar_productos"))
```

Con parámetros se pasan como kwargs:

```
return redirect(url_for("ver_producto", codigo=codigo))
```

> El primer argumento es **el nombre de la función Python**, no la URL. Si cambias la ruta del decorador, los `url_for` siguen funcionando.

El patrón "actúa → redirige"

Tras una acción que modifica estado (comprar, agregar, eliminar), **redirige** al acabar.

```
@app.route("/agregar/<codigo>/<nombre>/<float:precio>/<int:cantidad>")
def agregar(codigo, nombre, precio, cantidad):
    try:
        servicio.agregar_producto(codigo, nombre, precio, cantidad)
        return redirect(url_for("ver_producto", codigo=codigo))
    except ProductoYaExisteError as e:
        return str(e), 409
```

💡 **Por qué importa.**

Si el usuario pulsa *recargar*, se repite la última petición. Tras el redirect se repite la **consulta**, no la acción destructiva.

4. Estado de la aplicación en el servidor

Máquina de estados, usuarios concurrentes y persistencia

El `servicio` global

```
servicio = crear_servicio_sqlite()

app = Flask(__name__)

@app.route("/seleccionar/<codigo>")
def seleccionar(codigo):
    servicio.seleccionar(codigo)
    ...
```

- **Una sola instancia** atiende a todas las peticiones.
- El estado interno (`_saldo`, `_seleccion`) **se acumula** entre peticiones.
- Vive en memoria mientras el proceso Python esté corriendo.

Máquina de estados en el servidor

La expendedora es una máquina de estados: **el orden importa**.

```
GET /seleccionar/A1 → _seleccion = Item(A1)
GET /insertar/2.0 → _saldo += 2.0
GET /comprar → compra; reinicia estado
```

Cada route es una **transición**, no una transacción completa. El navegador no guarda nada; el servidor sí.

> Entre una petición y la siguiente pueden pasar segundos. El estado vive dentro del objeto `servicio`.

Problema 1: usuarios concurrentes

Dos navegadores apuntando al mismo servidor **se pisan el estado**.

```
Usuario A: /seleccionar/A1 _seleccion = A1
Usuario B: /seleccionar/B2 _seleccion = B2 ← pisa al A
Usuario A: /comprar → compra B2, no A1
```

⚠️ **El fallo no es del dominio.**

El dominio está diseñado para un usuario. El problema es asumir que ese único estado global sirve a cualquier cantidad de clientes.

Problema 2: estado perdido al reiniciar

Con diseño actual, el estado vive en la **RAM** del proceso. `Ctrl+C` → nuevo arranque → `_saldo = 0.0`.

Lo único que sobrevive: la **base de datos SQLite** con los productos.

💡 **Persistir en fichero JSON**

Guardar `_saldo` y `_seleccion` en `estado.json` tras cada cambio. Sobrevive al reinicio.

💡 **Sesiones HTTP (`flask.session`)**

Cada navegador recibe una cookie que identifica su propio estado. Resuelve el problema de concurrencia.

> Por ahora, la expendedora web es **monousuario y volátil**. Es honesto nombrarlo.

5. Parámetros en URL vs formularios HTML

Los límites de la URL y el concepto de *locale*

La URL no es una buena entrada de datos

```
/agregar_descuento/X1/Zumo/1.5/10/20.0
```

- **Legibilidad nula.** El orden es arbitrario; nadie adivina qué significa cada valor.
- **Caracteres especiales.** "Zumo de naranja" se convierte en `Zumo%20de%20naranja`.
- **Un error de orden** (porcentaje donde va cantidad) se acepta sin avisar.

⚠️ **Para pruebas vale. Para un usuario real, no.**
Ningún usuario escribe URLs de 6 parámetros en la barra del navegador.

El problema del separador decimal

En `es_ES` escribimos **coma**: `1,5`. Flask espera **punto**: `1.5`.

Locale: conjunto de convenciones culturales del sistema (idioma, formato de fechas, separador decimal, moneda). Se nombra con códigos como `es_ES`, `en_US`, `fr_FR`.

🚫 **Usuario español teclea**
`/agregar/X1/Zumo/1,5/10`
Obtiene un 404 misterioso. Desde su punto de vista "1,5" es un número válido.

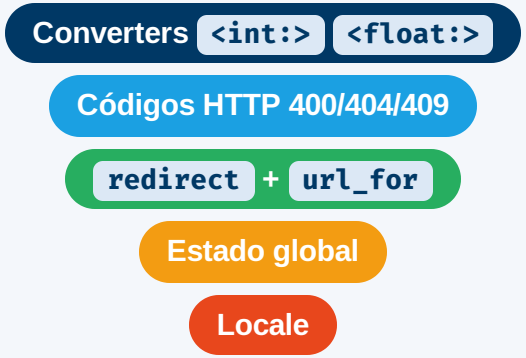
✅ **Flask usa siempre `\d+\.\d+`**
Porque es la convención de los lenguajes de programación, independiente del idioma del SO.

Hacia los formularios HTML (lab a5)

Aspecto	Parámetros en URL	Formularios HTML
Legibilidad	Baja	Alta: cada campo etiquetado
Caracteres especiales	Codificar manual	Lo hace el navegador
Separador decimal	Obligatorio `.`	`<input type="number">` lo abstrae
Apto para producción	Solo pruebas/lecturas	✅

> Usar URLs para **lecturas** (`/producto/A1`) es razonable. Para **altas** con muchos campos, es provisional. En el lab a5 lo sustituimos por `<form method="post">` .

¿Preguntas?



POO · CEPY · UT4 — Rutas dinámicas y excepciones en Flask