POO · CEPY · UT4

Manejadores, introspección y logging

Observabilidad y calidad en la interfaz web

Programación de aplicaciones en Python — Sesión 3

Índice

> Continuación de la sesión 2. Asume que ya conoces routes, converters, códigos HTTP y el estado global del servidor.

1. Manejadores globales de error

`@app.errorhandler` como red de seguridad

El problema que resuelven

Cada route captura sus excepciones de dominio con `try/except`. Pero hay errores **que no ocurren dentro de ningún route**:

- URL que no coincide con ninguna ruta registrada.
- Converter que rechaza un parámetro (`/reponer/A1/-1`).
- Excepciones no previstas en el código del route.
- Errores de infraestructura (BD caída, disco lleno...).

> ⚠️ **Sin manejo**
> Flask muestra una página genérica. En `debug=True`, el traceback técnico completo — útil al programador, confuso para el usuario.

El decorador `@app.errorhandler`

Una función decorada con `@app.errorhandler(codigo)` se ejecuta cuando Flask genera un error con ese código. Es un route especial que captura por **código HTTP** en vez de por URL (URL = `/producto/A1`; código HTTP = `404`).

```python
@app.errorhandler(404)
def no_encontrado(e):
    return f"<h2>404 — No encontrado</h2><p>{e}</p>", 404

@app.errorhandler(500)
def error_servidor(e):
    return "<h2>500 — Error del servidor</h2>", 500
```

> El parámetro `e` recibe el objeto de excepción que Flask ha generado (p.ej. `werkzeug.exceptions.NotFound` en un 404). Puedes interpolarlo, loguearlo o ignorarlo.

Interacción con los `try/except` del route

Los manejadores globales **no sustituyen** al `try/except`: lo **complementan**.

Situación	Responde
URL no registrada (`/foo`)	`@app.errorhandler(404)` global
Producto no encontrado capturado en route	`try/except` del route
Converter rechaza el valor (p.ej. `<int:>` con `tres`, o `/reponer/A1/-1`)	`@app.errorhandler(404)` global
Excepción no capturada en un route	`@app.errorhandler(500)` global

✅ **Los manejadores globales son una red de seguridad.**
Atrapan lo que se escapa de los `try/except`, no lo que ya está bien manejado.

Experiencia de usuario coherente

🚫 **Sin manejador global**

- URL inexistente → página Flask genérica
- Producto inexistente → tu mensaje personalizado

Dos experiencias distintas para el **mismo código HTTP 404**.

✅ **Con `@app.errorhandler(404)`**

- URL inexistente → tu manejador
- Converter rechazado → tu manejador
- Producto inexistente → tu mensaje del route

Apariencia coherente en todos los 404.

2. Introspección: la app conoce sus rutas

`app.url_map` y la página `/ayuda`

Qué es la introspección

Introspección es la capacidad de un programa de **observar su propio estado** en tiempo de ejecución.

> Como `dir()` y `type()` que ya usaste en Python, pero aplicado a la app.

Flask permite consultar qué rutas tiene registradas **sin leer el código fuente**: la información está viva dentro del objeto `app`.

> Aplicaciones: páginas de ayuda, documentación automática de APIs, depuración cuando un 404 aparece y no sabes por qué.

El objeto `app.url_map`

`app.url_map` es el mapa de rutas que Flask usa internamente. Se construye automáticamente con cada `@app.route( ... )`. Se itera con `iter_rules()`:

```
@app.route("/ayuda")
def ayuda():
    lineas = ["<h2>Rutas disponibles</h2><ul>"]
    for regla in app.url_map.iter_rules():
        if regla.endpoint ≠ "static":
            # acumular líneas y unirlas con join al final
            lineas.append(f"<li><code>{regla.rule}</code> — {regla.endpoint}</li>")
    lineas.append("</ul>")
    return "\n".join(lineas)
```

Atributo	Contiene	Ejemplo
`regla.rule`	El **patrón** tal como se declaró	`/producto/<codigo>`
`regla.endpoint`	Nombre de la función de vista	`ver_producto`

Patrón vs URL concreta

`regla.rule` muestra el **patrón**, no URLs concretas: verás `<codigo>` literal, no `A1` o `B2`.

```
/                          — inicio
/productos                 — listar_productos
/producto/<codigo>         — ver_producto
/reponer/<codigo>/<int:unidades>  — reponer
```

💡 **¿Por qué filtrar `static`?**

Flask registra automáticamente `/static/<path:filename>` para servir CSS, imágenes y JS. No forma parte de tu lógica de aplicación, por eso lo excluimos de `/ayuda`.

Una página de ayuda siempre al día

Página escrita a mano

- Hay que actualizarla cada vez que añades un route.
- Fácil olvidarlo.
- Rutas inventadas o desaparecidas conviven con las reales.

Generada por introspección

- Añades un route → aparece solo.
- Borras un route → desaparece solo.
- Imposible de desincronizar del código.

> La introspección es **la fuente de verdad**: el código y la documentación son el mismo dato.

3. Hooks y logging

Código transversal que no pertenece a ningún route

Qué es un hook

Un **hook** (literalmente *gancho*) es un punto del flujo de un programa donde puedes "enganchar" código propio para que se ejecute automáticamente, sin modificar el flujo original.

Flask ofrece varios hooks. El más útil para empezar: `@app.before_request`, que ejecuta una función **antes de cada petición**, pase por el route que pase.

> 💡 **Para qué sirven los hooks.**
> Añadir comportamientos transversales (logging, autenticación, medición de tiempos) sin tocar los routes uno por uno.

El módulo `logging`

Módulo estándar de Python para registrar mensajes. Ventajas sobre `print`:

- Escribe a **fichero** en vez de a consola.
- Clasifica mensajes por **nivel**: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`.
- Añade **timestamp** y contexto automáticamente.

```
import logging

logging.basicConfig(
    filename="expendedora.log",
    level=logging.INFO,
    # %(nombre)s es la sintaxis del módulo logging — no es f-string
    format="%(asctime)s [%(levelname)s] %(message)s",
)
```

`level=logging.INFO` captura INFO/WARNING/ERROR/CRITICAL. Descarta DEBUG.

`@app.before_request` y el objeto `request`

El hook recibe la petición actual a través de `request`, un objeto global que Flask **pone disponible automáticamente** dentro de cada petición:

```
from flask import request

@app.before_request
def log_peticion():
    app.logger.info(f"{request.method} {request.path}")
```

Propiedad	Contiene	Ejemplo
`request.method`	Método HTTP	`"GET"`
`request.path`	Ruta pedida (sin dominio)	`"/producto/A1"`

> Flask tiene su propio logger en `app.logger`. Usa la configuración de `basicConfig`.

Qué se versiona y qué no

Los ficheros de log son **generados en ejecución**: cambian cada vez que corres la app, dependen de qué URLs visites.

✅ Se versiona en Git

- El código que genera el log (`app.py` , `logging.basicConfig`).
- El `.gitignore` que excluye los logs.

🚫 No se versiona

- El propio `expendedora.log` .
- Bases de datos de desarrollo.
- Carpetas `__pycache__/` , `.venv/` .

```
## .gitignore
*.log
```

Mala práctica: `print` como sistema de registro

🚫 Incorrecto

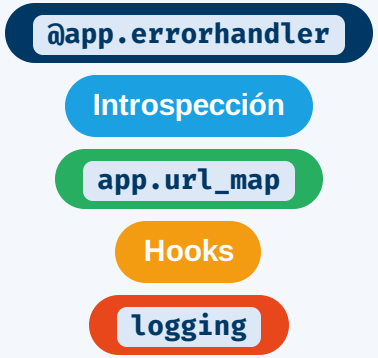
```python
@app.before_request
def log_peticion():
    print(f"{request.method} {request.path}")
```

- No queda traza al cerrar la terminal.
- No se puede filtrar por nivel.
- Sin timestamp ni formato.

✅ Correcto

```python
@app.before_request
def log_peticion():
    app.logger.info(f"{request.method} {request.path}")
```

¿Preguntas?



POO · CEPY · UT4 — Manejadores de error, introspección y logging