CEPY · Desarrollo en Python · UT4

Flask como nueva capa de presentación

Sesión 1 — Del menú de consola a la web

Programación Orientada a Objetos · Proyecto expendedora

Índice

1. Antes de empezar: los que haremos

Lo que suponemos antes de empezar con Flask

Anticipando

vamos a construir la interfaz web de vuestro proyecto personal con Flask.

Antes de que veamos qué es Flask:

Mirando el código de la expendedora.
¿Qué archivos/partes del proyecto creen que vamos a tener que **cambiar o reescribir** para añadir Flask?
¿Cuáles creen que vamos a quedar **exactamente igual**?

2. Qué es Flask y dónde encaja

La misma arquitectura, con un nuevo módulo en la capa de presentación

¿Qué es Flask?

Flask es un *microframework* web para Python.

- Asocia URLs a funciones Python normales.
- Cuando el navegador pide una URL, Flask llama a la función y envía su resultado como respuesta HTTP.

> A diferencia de frameworks pesados (Django), Flask **no impone estructura**: la arquitectura del proyecto se decide libremente. Por eso encaja con el diseño por capas que ya conocemos.

Arquitectura modular: núcleo + extensiones

El núcleo de Flask cubre lo imprescindible. Todo lo demás se añade como extensión opcional.

Funcionalidad	Cómo se obtiene
Rutas, peticiones, respuestas, plantillas	Núcleo de Flask
ORM (base de datos)	Flask-SQLAlchemy
Formularios con validación	Flask-WTF
Autenticación de usuarios	Flask-Login
Envío de correos	Flask-Mail

> **Instalamos solo lo que necesitamos.** En esta unidad usaremos el núcleo de Flask y Flask-WTF (en la sesión 3, cuando veamos formularios).

La arquitectura del proyecto expendedora

```
expendedora/
├── presentation/     ← menu.py (consola: print + input)
├── application/      ← servicios.py (fachada de casos de uso)
├── domain/           ← item.py, maquina.py, excepciones
└── infrastructure/   ← repositorio_sqlite.py, errores.py
```

La capa de presentación se ejecuta en `menu.py`. Habla con el servicio, muestra datos por consola y pide entrada con `input()`. Nada más.

Flask reemplaza `menu.py`

Antes

`menu.py` imprime con `print` y lee con `input`.

La aplicación es **standalone**: todo vive en un único proceso, en el mismo equipo, sin red.

Ahora

Flask atiende **peticiones HTTP** del navegador y responde con HTML.

La aplicación pasa a ser **cliente/servidor**: el navegador es el cliente, Flask el servidor.

El resto de capas (`application`, `domain`, `infrastructure`) **no cambia**.

Del menú a las URLs

En `menu.py`	En Flask
`print("...")`	HTML devuelto por la vista
`input("...")`	Petición del navegador (formulario — sesión 3)
`if opcion == "1":`	`@app.route('/productos')`
Bucle `while True:`	El servidor Flask (`app.run()`)
Una función por opción	Una función por ruta

La URL es la opción. La función es la acción. `@app.route` es lo que las conecta.

Los datos que el usuario introduce en la ruta se recogeráa través de un formulario — lo veremos en la sesión más adelante.

3. Primer código Flask

El `app.py` de la expendedora, línea a línea

Lo que ya sabemos: la cadena de construcción

En `presentation/menu.py`, antes del bucle del menú:

```
from expendedora.infrastructure.datos_iniciales import crear_servicio_sqlite
...

def main():
    servicio = crear_servicio_sqlite()
    while True:
        mostrar_menu()
        ...
```

`crear_servicio_sqlite()` construye la cadena `repo → maquina → servicio`. Hace **bootrasp** de la aplicación. No cambia en Flask.

`presentation/app.py` — la nueva capa de presentación

```
from flask import Flask
from expendedora.infrastructure.datos_iniciales import crear_servicio_sqlite
from expendedora.infrastructure.errores import ProductoNoEncontradoError

app = Flask(__name__)
servicio = crear_servicio_sqlite()


@app.route("/")
def inicio():
    return "<h1>Expendedora</h1>"


if __name__ == "__main__":
    app.run(debug=True)
```

Qué reconocemos aquí

- `crear_servicio_sqlite()` — función de inicio (*bootstrap*) que ya usa `menu.py`. Monta la cadena completa repo → máquina → servicio y devuelve un `ServicioExpendedora` listo para usar.
- `app = Flask(__name__)` — instancia de la clase `Flask`. OOP que ya conocemos.
- `@app.route("/")` — decorador que asocia una URL a una función.
- **La función de vista** - `inicio()` — función Python normal; lo que devuelve es lo que el navegador recibe.

> **El servicio, la máquina y el repositorio no saben si los llama `menu.py` o `app.py`.** Por eso podemos añadir Flask sin tocar ninguna de esas capas.

Una ruta, una acción

```
@app.route("/productos")
def listar_productos():
    return str(servicio.listar_productos())
```

El navegador pide `http://localhost:5000/productos`.

Flask busca el `@app.route` que coincide, llama a la función y devuelve su resultado.

El servicio **es el mismo**: `listar_productos()` se usaba igual desde el menú.

Rutas con parámetro variable

Cuando el servicio necesita parámetros se pasan en la URL

```
@app.route("/producto/<codigo>")
def ver_producto(codigo):
    producto = servicio.obtener_producto(codigo)
    return str(producto)
```

Llamadas y respuestas:

URL solicitada	Valor de `codigo`	Respuesta
`/producto/A1`	`"A1"`	Agua – 1.00 EUR
`/producto/D1`	`"D1"`	Refresco – 2.00 EUR
`/producto/ZZ`	`"ZZ"`	404: producto no existe

El try/except sigue existiendo... pero ¿dónde?

En `menu.py` lo teníamos así:

```
try:
    if opcion == "2":
        opcion_seleccionar(servicio)
    ...
except ProductoNoEncontradoError as e:
    print("X " + str(e))
```

La excepción llega desde el **dominio**. La captura la capa de **presentación**, porque es quien decide qué mostrarle al usuario.

En Flask, la capa de presentación es el route

```python
@app.route("/producto/<codigo>")
def ver_producto(codigo):
    try:
        producto = servicio.obtener_producto(codigo)
        return str(producto)
    except ProductoNoEncontradoError as e:
        return f"Error: {e}", 404
```

La estructura es idéntica a `menu.py`. Solo cambia lo que hacemos con el error: antes un `print`, ahora una respuesta HTTP con código 404. El servicio y el dominio **no saben nada** de esto.

4. Cierre y discusión

Lo que hemos visto, lo que viene ahora

Pregunta general

Si el route de Flask es la nueva capa de presentación...

¿Qué capas no deberían cambiar al añadir Flask?

¿Dónde gestionaremos los `try/except` de las excepciones de dominio?

Lo que haremos a continuación

Lab de sesión 1

Sobre el proyecto de la expendedora: crear `app.py`, construir la cadena de construcción y añadir rutas para listar y consultar productos, con manejo de `ProductoNoEncontradoError`.

¿Preguntas?


Capa de presentación
Cliente/servidor
@app.route
try/except en el route