POO · CEPY · UT4 · Fase 7

Autenticación y autorización

Login, sesión Flask y `@login_required` sin tocar el dominio

Curso de Especialización — Desarrollo de Aplicaciones con Python

Índice

Dónde estamos

Fase	Tema	Estado
1-5	Flask, rutas, errores, plantillas, formularios	✅
6	Mensajes flash + API REST	✅
7	**Autenticación y autorización**	**← AQUÍ**
8	Auditoría arquitectónica	—

🎯 **Promesa de esta fase.** Añadir login sin tocar `domain/` ni `ServicioExpendedora`. Otra vez la arquitectura por capas demuestra que las preocupaciones transversales caben sin reabrir el código del dominio.

1. Autenticación vs autorización

Dos preguntas que se encadenan pero son mecanismos independientes.

Las dos preguntas

🔑 Autenticación (AuthN)

"¿Quién eres?"

- Usuario presenta credenciales (nombre + contraseña)
- Servidor verifica contra la BD
- Pasa **una vez** al inicio
- Implementación: ruta `/login` + `check_password_hash`

🚦 Autorización (AuthZ)

"¿Puedes hacer esto?"

- Servidor mira si el usuario actual tiene permiso
- Pasa **en cada petición** a ruta protegida
- Implementación: decorador `@login_required` que consulta la sesión

⚠️ Se confunden porque el flujo típico **encadena** las dos. Pero son independientes: puedes autenticarte sin que ninguna ruta esté protegida, y proteger rutas sin distinguir usuarios.

2. Hash de contraseñas

Por qué no se guardan en claro y cómo se hashean con `werkzeug.security`.

`werkzeug.security` en dos llamadas

Las contraseñas **no se guardan nunca en claro**. Se guarda un **hash** unidireccional. Verificar consiste en volver a hashear lo que teclea el usuario y comparar.

```
from werkzeug.security import generate_password_hash, check_password_hash

password_hash = generate_password_hash('admin1234')     # al crear/cambiar
ok = check_password_hash(password_hash, 'admin1234')    # al verificar en /login
```

✅ **Viene con Flask.** Sin dependencia extra. `generate_password_hash` aplica salt aleatorio + algoritmo costoso por diseño.

🚫 **Mala práctica.** No uses `hash()` de Python (no es criptográfico) ni compares hashes con `==`. Toda la lógica criptográfica vive **dentro** de estas dos funciones.

3. Repositorio de usuarios paralelo

La decisión arquitectónica clave de la fase.

Por qué un repositorio nuevo

Los usuarios son **persistencia** pero **no son dominio**. La expendedora no sabe quién es su operario, igual que un microondas no sabe quién cocina.

```
domain/ intacto ✓
application/ intacto ✓
infrastructure/
├── repositorio_sqlite.py ← productos (ya existía)
└── repositorio_usuarios_sqlite.py ← NUEVO en a7
presentation/
└── app.py ← rutas /login,/logout + decorador
```

🚫 **Nunca** se añaden métodos de auth a `ServicioExpendedora`. La auth no es un caso de uso del dominio de venta. Indicador definitivo: `menu.py` (consola) sigue funcionando sin login y usa el mismo servicio.

> Misma BD `expendedora.db`, tabla nueva `usuarios`, fichero distinto.

Schema + repositorio

```
-- expendedora/crear_bd.py
CREATE TABLE IF NOT EXISTS usuarios (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nombre TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    rol TEXT NOT NULL DEFAULT 'admin',
    activo INTEGER NOT NULL DEFAULT 1
)
```

```
## expendedora/infrastructure/
##   repositorio_usuarios_sqlite.py

class RepositorioUsuariosSQLite:
    def __init__(self, ruta_bd):
        self._ruta_bd = ruta_bd

    def buscar_por_nombre(self, nombre):
        # SELECT ... WHERE nombre = ?
        # devuelve tupla o None

    def existe(self, nombre):
        return self.buscar_por_nombre(
            nombre) is not None
```

💡 `rol` y `activo` son **dato dormido** en esta fase. No se usan en el login, pero dejan el esquema preparado por si más adelante distingues admin/cliente o desactivas cuentas sin borrarlas.

4. Sesión Flask

Memoria entre peticiones del mismo navegador.

`session` vs `flash`

Mecanismo	Persistencia	Caso típico
`flash(mensaje)`	**De un uso** (se vacía al leer)	Feedback tras un POST con redirect
`session['clave']`	**Hasta `pop` o caducidad**	Recordar que el usuario está logueado

Las dos comparten infraestructura: la misma **cookie firmada** con `app.secret_key`. Si ya declaraste `app.secret_key` para los flashes en a6, no hay nada nuevo que configurar.

⚠️ **Firmada, no cifrada.** El contenido de la cookie viaja **legible** — cualquiera con DevTools del navegador puede leerlo. En `session` va solo el **identificador** del usuario, nunca contraseñas o datos sensibles.

API de `session`

```python
from flask import session

session['user'] = 'admin'           # guardar (login)
nombre = session.get('user')        # leer (devuelve None si no existe)
'user' in session                   # comprobar
session.pop('user', None)           # eliminar (logout) — None evita KeyError
session.clear()                     # vaciar todo
```

💡 `session` se comporta como un **diccionario normal** en Python — `get`, `pop`, `clear`, `in` funcionan igual.

✅ En plantillas Jinja, `session` ya está disponible. `{% if session['user'] %}` **no levanta KeyError** — devuelve `Undefined` (falso) si la clave no está.

5. El decorador `@login_required`

Decoradores Python aplicados a protección de rutas.

Decoradores Python — repaso visual

```python
def mi_decorador(f):
    def envoltorio(*args, **kwargs):
        print(f"Antes de {f.__name__}")
        resultado = f(*args, **kwargs)
        print(f"Después de {f.__name__}")
        return resultado
    return envoltorio

@mi_decorador
def saludar(nombre):
    print(f"Hola, {nombre}")

saludar('Ana')
```

```
Antes de saludar
Hola, Ana
Después de saludar
```

`@mi_decorador` encima de `def saludar` es la forma corta de `saludar = mi_decorador(saludar)`. Después de esa línea, `saludar` **ya no es** la función original — es el `envoltorio` devuelto por el decorador.

Anatomía de `@login_required`

```python
## expendedora/presentation/app.py
from functools import wraps

def login_required(f):
    @wraps(f)
    def envoltorio(*args, **kwargs):
        if 'user' not in session:
            flash('Necesitas iniciar sesión para esa acción.', 'error')
            return redirect(url_for('login', siguiente=request.path))
        return f(*args, **kwargs)
    return envoltorio
```

⚠️ **`@wraps(f)` es obligatorio.** Copia el nombre de `f` al envoltorio. Sin él, Flask vería que todas las vistas decoradas se llaman `envoltorio`, colisionarían en el `url_map`, y `url_for('agregar')` daría `BuildError`.

El parámetro `siguiente`

```
GET /agregar (sin sesión)
└→ @login_required → redirect(/login?siguiente=/agregar)
└→ GET /login?siguiente=/agregar (formulario)
└→ POST /login?siguiente=/agregar (admin/admin OK)
└→ redirect(/agregar) ← respeta siguiente
```

El decorador **pone** `siguiente=request.path` al redirigir. La rama POST de `/login` lo **lee** con `request.args.get('siguiente')` y redirige ahí en lugar de a inicio.

🚫 El `<form method="POST">` de `login.html` **no lleva** atributo `action`. Si lo tuviera (`action="/login"`), el querystring `?siguiente=...` se perdería al enviar y el usuario aterrizaría siempre en la raíz, ignorando el destino original.

6. Orden de los decoradores

Una regla técnica que evita un fallo oculto de seguridad.

Correcto vs incorrecto

Los decoradores se aplican **de abajo arriba**: `@app.route` registra lo que tiene **debajo** en ese momento — la función pelada o un envoltorio ya creado.

✅ Correcto

```
@app.route('/agregar', methods=['POST'])
@login_required
def agregar():
    ...
```

1. `@login_required` envuelve `agregar` → `envoltorio`; `@app.route` registra `envoltorio` en el `url_map`
2. Flask llama a `envoltorio`, que comprueba sesión ✓

🚫 Incorrecto

```
@login_required
@app.route('/agregar', methods=['POST'])
def agregar():
    ...
```

1. `@app.route` registra `agregar` **pelada**; luego `@login_required` envuelve, pero Flask ya guardó la versión sin proteger
2. **Decorador colgado.** Ruta accesible sin sesión. **Fallo silencioso.**

7. Autorización por rol funcional

Qué proteger no es una decisión de HTTP. Es una decisión de dominio.

Cliente vs operario

Tipo de actor	Rutas	Protección
Cliente (uso normal de la máquina)	`/`, `/productos`, `/producto/<codigo>`, `/buscar`, `/saldo`, `/insertar`, `/seleccionar`, `/comprar`, `/cancelar`, `/api/*`	⭕ libre
Operario (mantenimiento del catálogo)	`/agregar`, `/eliminar/<codigo>`, `/reponer/<codigo>/<int:unidades>`	🔒 `@login_required`

🚫 **Mala práctica: proteger por verbo HTTP.** Tanto `/comprar` (cliente) como `/agregar` (operario) son POSTs que mutan estado. Si decidieras "todo lo que muta lleva login", bloquearías `/comprar` que es legítimamente anónima.

> Si exigieras login para comprar, la máquina dejaría de ser una **expendedora** y se convertiría en una **tienda online** — un producto distinto, con casos de uso distintos. La pregunta es siempre *¿quién es el actor legítimo de esta acción?*

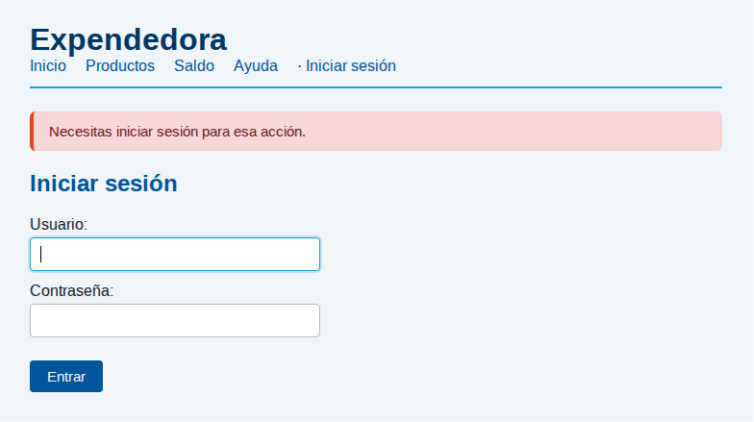
8. El flujo completo

Login + acceso protegido + logout, paso a paso con capturas reales.

Login — paso 1-2: redirect al login

El usuario sin sesión intenta acceder a una ruta protegida:

```
GET /agregar
   ↓
@login_required detecta no-sesión
   ↓
flash('Necesitas iniciar sesión ... ')
   ↓
redirect('/login?siguiente=/agregar')
```



💡 El parámetro `siguiente` se mete en la URL **aquí**, en el redirect del decorador. Es lo que permitirá volver a `/agregar` tras autenticarse.

Login — paso 3-4: formulario vacío

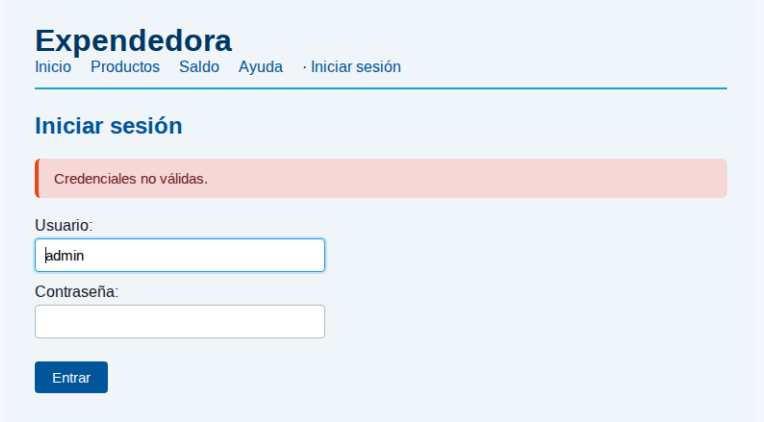
El navegador sigue el redirect. La rama GET de `/login` renderiza el formulario:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        ...
    return render_template(
        'login.html',
        error=None,
        nombre=''
    )
```

⚠️ El `<form method="POST">` de `login.html` **NO lleva** `action`. Al enviar, el navegador usa la URL actual y conserva el `?siguiente= ...`.

Login — paso 5: credenciales no válidas

```
if request.method == 'POST':
    nombre = request.form.get('nombre', '').strip()
    password = request.form.get('password', '')
    # fila: (id, nombre, password_hash, rol, activo)
    fila = repositorio_usuarios.buscar_por_nombre(nombre)
    if fila and fila[4] == 1 \
       and check_password_hash(fila[2], password):
        session['user'] = fila[1]
        return redirect( ... )
    return render_template(
        'login.html',
        error='Credenciales no válidas.',
        nombre=nombre
    ), 401
```



⚠️ Mensaje **idéntico** tanto si el usuario no existe como si la contraseña falla. No revela qué cuentas son válidas.

Login — paso 5 OK + paso 7: sesión activa

```
if fila and fila[4] == 1 \
   and check_password_hash(fila[2], password):
    session['user'] = fila[1]
    flash(f"Bienvenido, {fila[1]}.", 'exito')
    destino = request.args.get('siguiente') \
              or url_for('inicio')
    return redirect(destino)
```

Lo importante:

- `session['user']` guarda el identificador
- `request.args.get('siguiente')` lee el destino original
- `redirect(destino)` cierra el ciclo

✅ Aquí termina la autenticación. La sesión vive en la cookie hasta `pop` o caducidad.

Logout — paso 1: botón visible en la nav

```
<!— expendedora/presentation/templates/base.html —>
<nav>
  <a href="{{ url_for('listar_productos') }}">
    Productos</a> ·
  …
  {% if session['user'] %}
    · <a href="{{ url_for('agregar') }}">Agregar</a>
    · <span>Sesión: {{ session['user'] }}</span>
    · <form method="POST"
            action="{{ url_for('logout') }}">
        <button>Cerrar sesión</button>
      </form>
  {% else %}
    · <a href="{{ url_for('login') }}">Iniciar sesión</a>
  {% endif %}
</nav>
```

🚫 **Logout va por POST, no por enlace `<a>`.** GET no muta. Un enlace lo dispararía cualquier prefetch del navegador o un `<img src="/logout">` malicioso.

Logout — paso 2-4: sesión cerrada

```python
## expendedora/presentation/app.py

@app.route('/logout', methods=['POST'])
def logout():
    session.pop('user', None)
    flash('Sesión cerrada.', 'info')
    return redirect(url_for('inicio'))
```

Tres líneas que cierran el ciclo:

- `session.pop('user', None)` — elimina la clave (con `None` evita `KeyError` si la cookie ya no tenía sesión)
- `flash( ... )` — mensaje de despedida en la siguiente página
- `redirect( ... )` — fuerza un GET nuevo; la nav se renderiza ya sin sesión

¿Preguntas?

Entregable: `e_6` → carpeta `05-flask-06`, versión `0.11.0`

Próxima fase: auditoría arquitectónica del proyecto completo