Showcase AOP en Admin Dashboard

Un ejemplo práctico con Flask y Decoradores

Equipo AOP

Proyecto de Tópicos de Ingeniería de Software III

June 21, 2025

¿Qué es Programación Orientada a Aspectos (AOP)?

- Es un paradigma de programación que busca la **separación de incumbencias** (separation of concerns).

¿Qué es Programación Orientada a Aspectos (AOP)?

- Es un paradigma de programación que busca la **separación de incumbencias** (separation of concerns).
- Permite modularizar funcionalidades que "atraviesan" múltiples partes de una aplicación, conocidas como *cross-cutting concerns*.

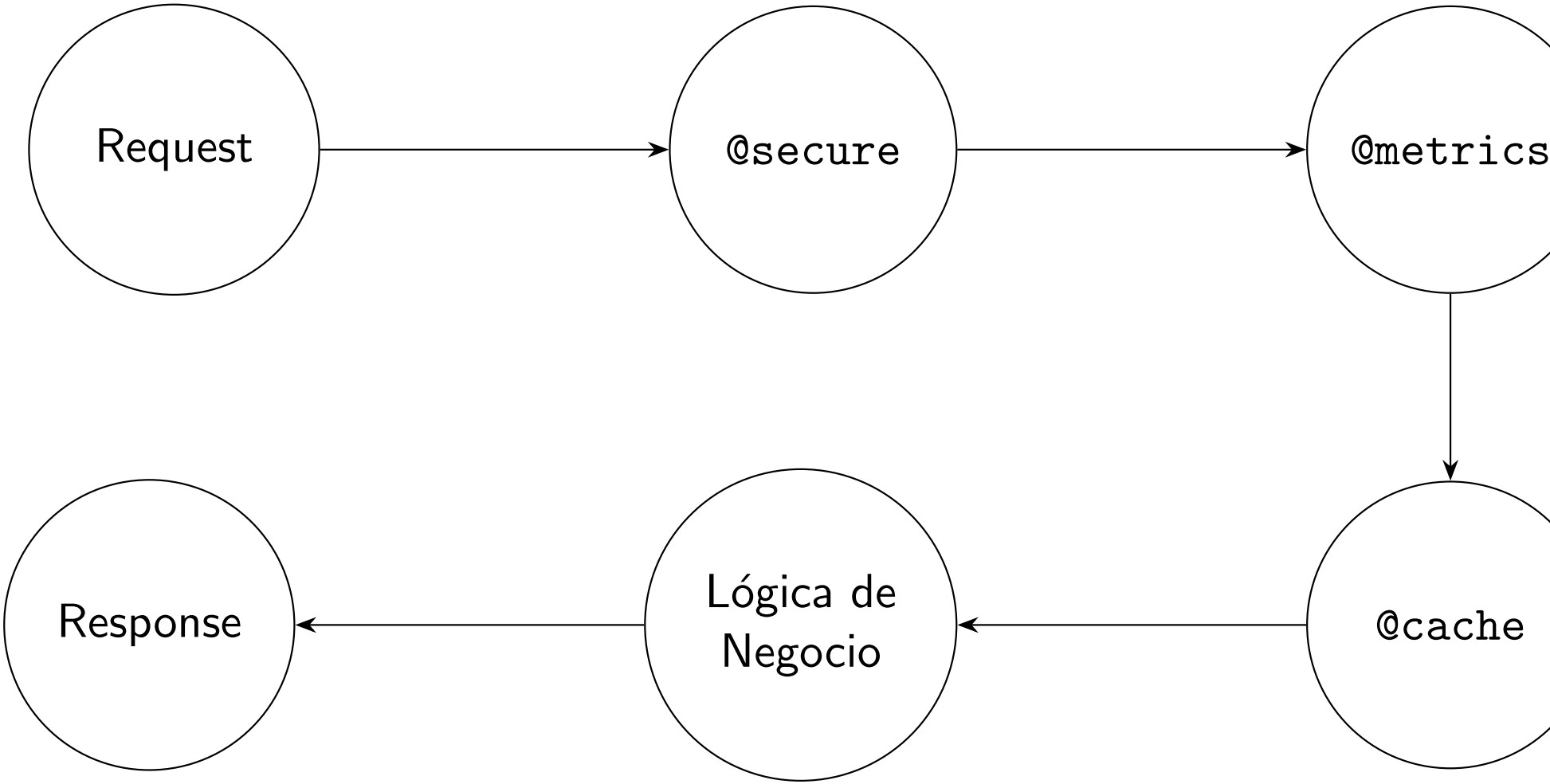
¿Qué es Programación Orientada a Aspectos (AOP)?

- Es un paradigma de programación que busca la **separación de incumbencias** (separation of concerns).
- Permite modularizar funcionalidades que "atraviesan" múltiples partes de una aplicación, conocidas como *cross-cutting concerns*.
- Ejemplos comunes:
  - Logging y Auditoría
  - Seguridad y Autenticación
  - Caching
  - Medición de rendimiento (Métricas)

¿Qué es Programación Orientada a Aspectos (AOP)?

- Es un paradigma de programación que busca la **separación de incumbencias** (separation of concerns).
- Permite modularizar funcionalidades que "atraviesan" múltiples partes de una aplicación, conocidas como *cross-cutting concerns*.
- Ejemplos comunes:
  - Logging y Auditoría
  - Seguridad y Autenticación
  - Caching
  - Medición de rendimiento (Métricas)
- En Python, una forma muy popular de implementar AOP es a través de **decoradores**.

Flujo de un Request a través de los Aspectos



- Un request HTTP es interceptado secuencialmente por cada decorador.
- Cada "aspecto" añade su comportamiento *antes* o *después* de ejecutar la lógica principal de la vista.

Aspecto de Seguridad: `@secure`

Objetivo

Proteger rutas para que solo usuarios con roles específicos puedan acceder.

- Se implementa como un decorador que recibe una lista de roles permitidos.
- Verifica el rol del usuario almacenado en la sesión (‘g.user‘).
- Si el usuario no tiene el rol adecuado, aborta la petición con un error HTTP 403 (Forbidden).
- Si no está logueado, se le redirige al login (manejado por `@login_required`).

Implementación del Decorador @secure

```python
def secure(roles=None):
    if roles is None:
        roles = []

    def decorator(view):
        @functools.wraps(view)
        def wrapped_view(**kwargs):
            # Asumimos que g.user existe
            if g.user is None:
                abort(401)  # Unauthorized

            if roles and g.user['role'] not in roles:
                abort(403)  # Forbidden

            # Log de seguridad (otro aspecto)
            print(f"Acceso concedido a {g.user['username']}"
    )
            return view(**kwargs)
        return wrapped_view
    return decorator

```

Aspectos de Cache y Métricas

Objetivos

- `@cache(ttl=60)`: Almacena el resultado de una vista en memoria para evitar recálculos costosos. El 'ttl' (Time To Live) define la duración en segundos.
- `@metrics`: Mide y reporta el tiempo de ejecución de una vista. Es útil para identificar cuellos de botella.

Orden de Aplicación

El orden de los decoradores importa. `@metrics` debe envolver a `@cache` para medir el tiempo real, incluyendo el acierto de caché.

Ejemplo de Uso en la Vista de Productos

```
1  @bp.route('/')
2  @login_required
3  @metrics
4  @cache(ttl=60)
5  def list_products():
6      """Muestra la lista de productos."""
7      db = get_db()
8      products = db.execute(
9          'SELECT id, name, price, stock FROM product'
10     ).fetchall()
11     return render_template('fragments/products.html',
12                            products=products)
13
```

Resultados en Consola

Primer request (Cache Miss):

```
1 METRICS for 'list_products':
2   - Execution Time: 0.0028s
3 CACHE: Miss for key 'list_products:{}'. Caching result.
4
```

Segundo request (Cache Hit):

```
1 METRICS for 'list_products':
2   - Execution Time: 0.0001s
3 CACHE: Hit for key 'list_products:{}'.
4
```

Auditoría (@audit) y Feature Flags (@feature_flag)

@audit(action='view_order')

Registra quién, qué y cuándo se realizó una acción. Fundamental para trazabilidad.

```
@bp.route('/<int:id>')
@login_required
@audit(action='view_order')
def view_order(id):
    # ...

```

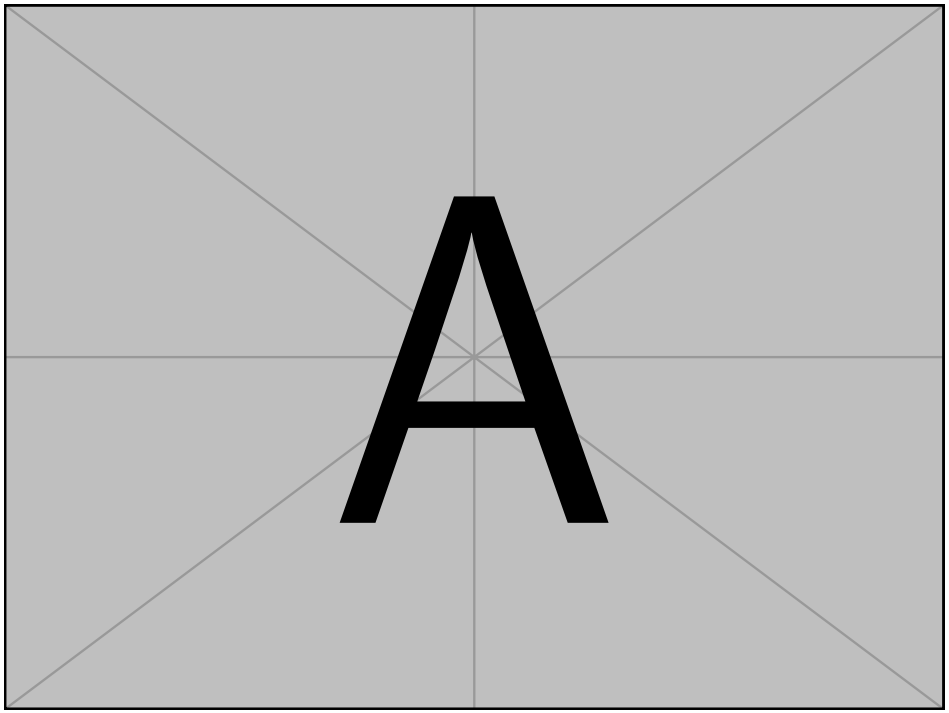
@feature_flag('promo_editor')

Permite activar o desactivar funcionalidades en tiempo real sin redesplegar. La UI se adapta según un valor en la BBDD.

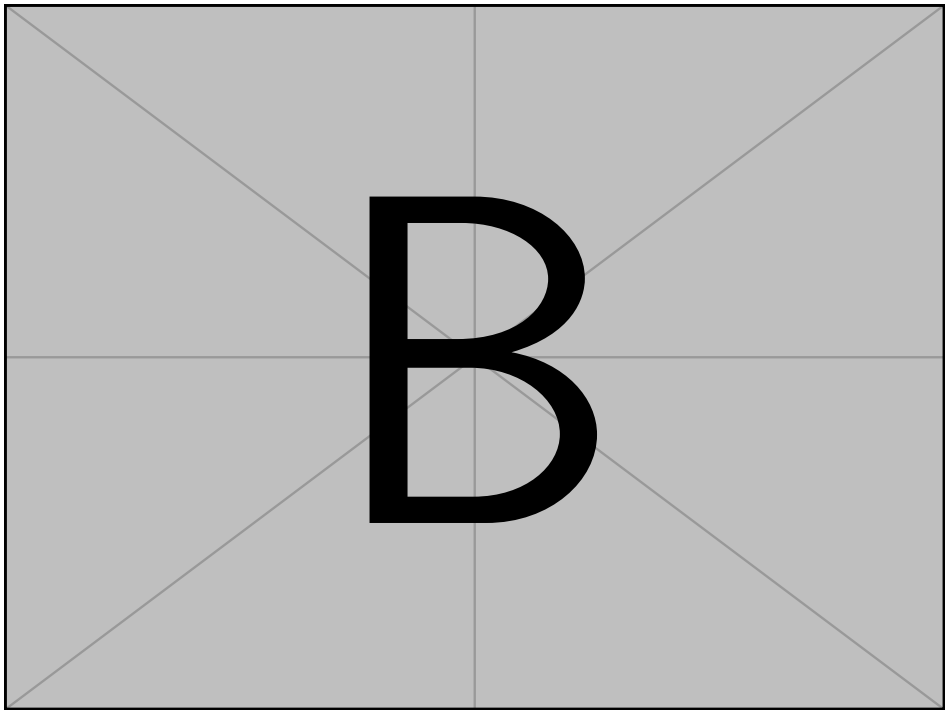
```
@bp.route('/', methods=('GET', 'POST'))
@login_required
@feature_flag('promo_editor')
def manage_promotions():
```

Demo: UI del Dashboard

Login de Usuario



Lista de Productos



Nota: Reemplazar con capturas de pantalla reales de la aplicación.

Cómo Ejecutar el Demo Local

Pasos

1. Clonar el repositorio y crear entorno virtual.
2. Instalar dependencias:

```
pip install -r requirements.txt

```

3. Inicializar la base de datos (crea y puebla ‘project.sqlite‘):

```
flask --app app init-db

```

4. Ejecutar la aplicación:

```
flask --app run

```

5. Acceder en `http://127.0.0.1:5000` (user: admin, pass: admin).

Ventajas y Conclusiones

- **Código más limpio:** La lógica de negocio (vistas de Flask) no se contamina con código de seguridad, cache, etc.

Conclusión Final

AOP, a través de decoradores en Python, es una herramienta poderosa para construir aplicaciones robustas, modulares y fáciles de mantener.

Ventajas y Conclusiones

- **Código más limpio:** La lógica de negocio (vistas de Flask) no se contamina con código de seguridad, cache, etc.
- **Reutilización:** Los decoradores se pueden aplicar a cualquier vista con una sola línea.

Conclusión Final

AOP, a través de decoradores en Python, es una herramienta poderosa para construir aplicaciones robustas, modulares y fáciles de mantener.

Ventajas y Conclusiones

- **Código más limpio:** La lógica de negocio (vistas de Flask) no se contamina con código de seguridad, cache, etc.
- **Reutilización:** Los decoradores se pueden aplicar a cualquier vista con una sola línea.
- **Mantenibilidad:** Si se necesita cambiar la lógica de logging, solo se modifica el decorador `@audit`, no todas las vistas que lo usan.

Conclusión Final

AOP, a través de decoradores en Python, es una herramienta poderosa para construir aplicaciones robustas, modulares y fáciles de mantener.

Ventajas y Conclusiones

- **Código más limpio:** La lógica de negocio (vistas de Flask) no se contamina con código de seguridad, cache, etc.
- **Reutilización:** Los decoradores se pueden aplicar a cualquier vista con una sola línea.
- **Mantenibilidad:** Si se necesita cambiar la lógica de logging, solo se modifica el decorador `@audit`, no todas las vistas que lo usan.
- **Testabilidad:** Cada aspecto puede ser probado de forma aislada.

Conclusión Final

AOP, a través de decoradores en Python, es una herramienta poderosa para construir aplicaciones robustas, modulares y fáciles de mantener.

¿Preguntas?