Showcase AOP en Admin Dashboard

Un ejemplo práctico con Flask y Decoradores

Equipo AOP

Proyecto de Tópicos de Ingeniería de Software III

23 de junio de 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*.

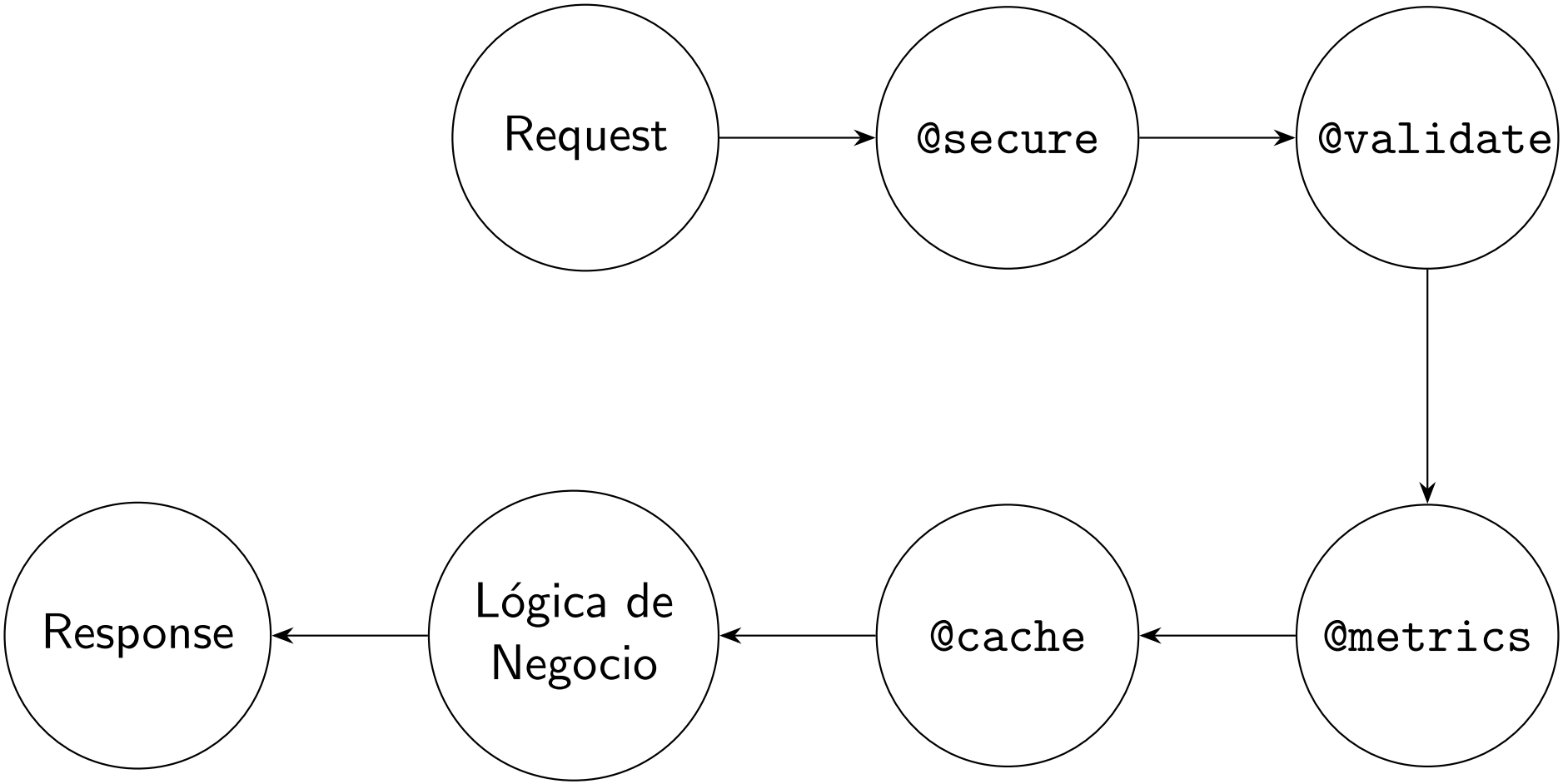
¿Qué es Programación Orientada a Aspectos (AOP)?

- 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 y Métricas
  - **Validación de datos**

¿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 y Métricas
  - **Validación de datos**
- 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):
            if g.user is None:
                abort(401)  # Unauthorized

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

            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 @metrics
3 @cache(ttl=60)
4 def list_products():
5     # ...
6
```

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.

```
1 @audit(action='view_order')
2 def view_order(id): # ...
3
```

@feature_flag('promo_editor')

Permite activar o desactivar funcionalidades en tiempo real.

```
1 @feature_flag('promo_editor')
2 def manage_promotions(): # ...
3
```

Aspecto de Validación: `@validate_with`

Objetivo

Validar los datos de entrada (ej. formularios) de forma declarativa, separando las reglas de la lógica de la vista.

- Se define un **esquema** con Pydantic que representa la estructura y reglas de los datos.
- El decorador `@validate_with` intercepta el request, valida los datos contra el esquema y, si es exitoso, los adjunta al objeto ‘g‘ de Flask.
- Si la validación falla, muestra un error al usuario y detiene la ejecución.

Paso 1: Definir el Esquema Pydantic

```
1  // file: app/schemas.py
2  from pydantic import BaseModel, Field
3  from datetime import date
4
5  class PromotionSchema(BaseModel):
6      name: str
7      discount_percent: float = Field(
8          ..., gt=0, lt=100,
9          description="El descuento debe ser entre 0 y 100."
10     )
11     start_date: date
12     end_date: date
13
```

Paso 2: Aplicar el Decorador

```
1  // file: app/promotions/routes.py
2  from app.schemas import PromotionSchema
3
4  @bp.route('/', methods=('GET', 'POST'))
5  @feature_flag('promo_editor')
6  @validate_with(PromotionSchema)
7  def manage_promotions():
8      if request.method == 'POST':
9          # No hay if/else de validacion!
10         # Los datos ya estan validados en g.validated_data
11         promo = g.validated_data
12         db.execute(
13             'INSERT INTO promotion ...',
14             (promo.name, promo.discount_percent, ...)
15         )
16         # ...
17
```

Demo: UI del Dashboard

Login de Usuario

Lista de Productos

Nota: Las imágenes son representativas del UI del dashboard.

Cómo Ejecutar el Demo Local

Pasos

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

```
uv pip install -r requirements.txt

```

3. Instalar el proyecto en modo editable (para que pytest funcione):

```
pip install -e .

```

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

```
flask --app app init-db

```

5. Ejecutar la aplicación:

```
flask --app app run

```

Ventajas y Conclusiones

- **Código más limpio:** La lógica de negocio no se contamina con código de seguridad, cache, validación, 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 no se contamina con código de seguridad, cache, validación, 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 no se contamina con código de seguridad, cache, validación, 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 no se contamina con código de seguridad, cache, validación, 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?