Diseño de aplicaciones: Patrón Model-View-Controller (MVC)
© Universitas Europaea IMF

Indice

Diseño de aplicaciones: Patrón Model-View-Controller (MVC)

1. Introducción al Patrón Model-View-Controller (MVC)

1.1. ¿Qué es el patrón MVC?

El **patrón Model-View-Controller (MVC)** es una arquitectura de software que separa la lógica de la aplicación en tres componentes principales: **Modelo (Model), Vista (View) y Controlador (Controller)**. Su objetivo es mejorar la organización del código, facilitar el mantenimiento y promover la reutilización en el desarrollo de aplicaciones. El patrón MVC es una arquitectura fundamental en el desarrollo de software moderno, especialmente en aplicaciones web. Su enfoque modular y la separación de responsabilidades mejoran la **escalabilidad, reutilización y mantenimiento** del código, facilitando la colaboración entre diferentes equipos de desarrollo.

Definición y Origen: El patrón MVC fue introducido en la década de 1970 por **Trygve Reenskaug**, mientras trabajaba en el laboratorio Xerox PARC. Originalmente, fue diseñado para mejorar la interacción entre usuarios y sistemas de software en entornos gráficos.

A lo largo del tiempo, MVC se ha convertido en una referencia en el desarrollo de aplicaciones web y de escritorio, siendo adoptado en frameworks como **Django, Flask, Ruby on Rails y Spring**.

MVC se estructura en tres componentes:

- **Modelo (Model):** Gestiona los datos, lógica de negocio y acceso a bases de datos.
- **Vista (View):** Se encarga de la presentación y la interfaz con el usuario.
- **Controlador (Controller):** Actúa como intermediario entre el modelo y la vista, gestionando la lógica de control y las interacciones del usuario.

Beneficios de la Separación de Responsabilidades:

La estructura MVC ofrece múltiples ventajas al separar la lógica de la aplicación en diferentes capas:

Mantenimiento y escalabilidad:

- Facilita la actualización de componentes sin afectar el resto del sistema.
- Permite agregar nuevas funcionalidades con menor impacto en el código existente.

Reutilización de código:

- Las vistas pueden reutilizarse con diferentes modelos sin necesidad de modificar la lógica de negocio.
- Los controladores pueden manejar múltiples vistas sin duplicar código.

Facilidad para la colaboración en equipos:

- Los desarrolladores backend pueden trabajar en el **Modelo** sin interferir con el desarrollo de la **Vista**.
- Los diseñadores pueden modificar la apariencia del sistema sin afectar la lógica del negocio.

Mejor organización del código:

- La separación de responsabilidades reduce el acoplamiento y mejora la claridad del código.
- Hace que el software sea más fácil de entender y depurar.

1.2. Componentes del MVC

E l **patrón Model-View-Controller (MVC)** divide la estructura de una aplicación en tres componentes principales: **Modelo (Model), Vista (View) y Controlador (Controller)**. Esta separación de responsabilidades permite un código más organizado, modular y escalable, facilitando el mantenimiento y la reutilización de componentes. El **Modelo** gestiona los datos y la lógica de negocio, la **Vista** presenta la información al usuario y el **Controlador** coordina la comunicación entre ambos. Esta separación permite un desarrollo más modular, facilitando la escalabilidad y mantenimiento del software

Modelo (Gestión de Datos y Lógica de Negocio):

El **Modelo** es la capa encargada de manejar los datos, la lógica de negocio y la comunicación con la base de datos. Define cómo se almacenan, recuperan y procesan los datos dentro del sistema.

Responsabilidades del Modelo:

- Gestionar la estructura de los datos y su validación.
- Implementar reglas de negocio, como cálculos, validaciones y restricciones.
- Comunicarse con la base de datos o APIs externas para recuperar o guardar información.

Ejemplo en Python (Flask/Django):

```
class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    def calcular_descuento(self, porcentaje):
        return self.precio - (self.precio * porcentaje / 100)
```

Este **modelo** define un producto con un nombre y un precio, además de un método para calcular descuentos.

Vista (Presentación de la Información)

L a **Vista** es la capa encargada de mostrar la información al usuario. Se encarga de renderizar datos provenientes del modelo y definir la interfaz gráfica o la estructura de salida en una aplicación web.

Responsabilidades de la Vista:

- Presentar los datos al usuario de forma clara y estructurada.
- Capturar la interacción del usuario mediante formularios o eventos.
- Renderizar páginas HTML en aplicaciones web o interfaces gráficas en aplicaciones de escritorio.

Ejemplo en Flask con una plantilla HTML (Jinja2):

```
<!DOCTYPE html>
<html>
<head>
   <title>Lista de Productos</title>
</head>
<body>
   <h1>Productos Disponibles</h1>
   <ul>
      {% for producto in productos %}
         <li>{{ producto.nombre }} - ${{ producto.precio }}</li>
      {% endfor %}
   </ul>
</body>
</html>
```

Aquí, la **vista** muestra una lista de productos obtenidos desde el modelo.

Controlador (Coordinación entre el Modelo y la Vista):

El **Controlador** es el intermediario que gestiona la comunicación entre la Vista y el Modelo. Se encarga de procesar las solicitudes del usuario, llamar a los métodos adecuados en el modelo y enviar la información a la vista para su presentación.

Responsabilidades del Controlador:

- Recibir las solicitudes del usuario.
- Llamar a los métodos adecuados en el modelo para procesar la información.
- Pasar los datos a la vista para ser presentados.

Ejemplo en Flask:

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def mostrar_productos():
    productos = [Producto("Laptop", 1200), Producto("Teléfono", 800)]
    return render_template("productos.html", productos=productos)

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

En este caso, el **controlador** (mostrar_productos()) obtiene los datos del modelo y los pasa a la vista para su renderización.

1.3. Comparación entre MVC y otros patrones arquitectónicos

El patrón **Model-View-Controller (MVC)** es ampliamente utilizado en el desarrollo de software debido a su capacidad para separar responsabilidades y mejorar la mantenibilidad del código. Sin embargo, existen otros enfoques arquitectónicos, como el **patrón de diseño en capas** y las **arquitecturas monolíticas sin separación clara**, con los que MVC presenta diferencias significativas. El **MVC** es un patrón ideal para aplicaciones interactivas, ya que mejora la organización del código y permite un desarrollo más modular. Aunque el **diseño en capas** puede ser más adecuado para sistemas grandes con múltiples niveles de procesamiento, **MVC es más eficiente que una arquitectura monolítica**, asegurando un software más escalable y mantenible.

Diferencias con el Patrón de Diseño en Capas: El **patrón de diseño en capas** organiza el software en diferentes niveles, donde cada capa tiene una función específica y se comunica con la capa inmediata. Las capas más comunes son:

- **Capa de Presentación:** Maneja la interfaz con el usuario.
- **Capa de Lógica de Negocio:** Procesa reglas de negocio y validaciones.
- **Capa de Acceso a Datos:** Gestiona la comunicación con bases de datos.

Diferencias clave entre MVC y el diseño en capas:

Característica	MVC	Diseño en Capas
Estructura	Basado en Modelo, Vista y Controlador.	Separa el software en múltiples capas.
Flujo de datos	Controlador dirige el flujo de información entre Modelo y Vista.	Las capas se comunican de manera secuencial.
Flexibilidad	Más orientado a la interacción con el usuario.	Más útil en aplicaciones grandes con lógica de negocio compleja.
Uso común	Aplicaciones web y de escritorio.	Sistemas empresariales y arquitecturas distribuidas.

Aunque MVC puede considerarse un caso particular de arquitectura en capas, su enfoque en la separación clara entre la lógica de negocio y la interfaz de usuario lo hace más adecuado para aplicaciones interactivas.

Ventajas frente a Arquitecturas Monolíticas sin Separación Clara: En una **arquitectura monolítica tradicional**, el código de la aplicación suele estar altamente acoplado, sin una división clara entre la lógica de negocio y la interfaz de usuario. Esto puede generar problemas como:

- **Dificultad en el mantenimiento:** Modificar una parte del sistema puede afectar otras sin querer.
- **Código desorganizado:** Mezclar lógica de negocio con interfaz de usuario dificulta la escalabilidad.
- **Problemas de reutilización:** No se pueden reutilizar componentes sin modificar varias partes del código.

En contraste, **MVC** ofrece las siguientes ventajas:

- **Mejor mantenimiento:** Separar la lógica del negocio de la vista facilita modificaciones sin afectar todo el sistema.
- **Mayor escalabilidad:** Se pueden agregar nuevas funcionalidades sin afectar las capas existentes.
- **Código más estructurado:** Cada componente tiene una función específica, reduciendo la complejidad.

2. Implementación Práctica del MVC en Python

2.1. Introducción a Frameworks Python para MVC

El patrón **Model-View-Controller (MVC)** se implementa en Python mediante distintos frameworks que facilitan la estructuración del código y la gestión de aplicaciones web. Dos de los frameworks más utilizados para aplicar MVC en Python son **Flask** y **Django**, cada uno con enfoques y características diferentes.

Flask es ideal para proyectos pequeños y flexibles, permitiendo un desarrollo rápido con mínima configuración mientras que **Django** ofrece un enfoque estructurado y robusto, adecuado para aplicaciones empresariales escalables. La elección entre Flask y Django dependerá del tamaño, complejidad y necesidades del proyecto.

2.2. Flask: Enfoque Ligero para Aplicaciones Pequeñas

Flask es un framework minimalista y flexible que permite desarrollar aplicaciones web de manera rápida y sencilla. Su diseño modular permite que el desarrollador elija las herramientas que mejor se adapten a su proyecto, sin imponer una estructura rígida.

Características principales:

- **Ligereza:** Flask no impone configuraciones predefinidas, lo que permite desarrollar aplicaciones con solo los componentes esenciales.
- **Extensibilidad:** Se pueden agregar extensiones para manejo de bases de datos, autenticación y otras funcionalidades según sea necesario.
- **Facilidad de aprendizaje:** Su sintaxis simple y su documentación clara lo hacen ideal para principiantes.

Ejemplo de estructura MVC en Flask:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')

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

Cuándo usar Flask:

- Para **proyectos pequeños o medianos**.
- Aplicaciones web que requieren **rápida implementación y flexibilidad**.
- APIs REST o servicios web con bajo consumo de recursos.

2.3. Django: Enfoque Estructurado con Herramientas

Integradas:

Django es un framework robusto que sigue el principio **"Batteries included"**, lo que significa que proporciona herramientas integradas para manejar bases de datos, autenticación, seguridad y administración.

Características principales:

- **Estructura predefinida:** Organiza automáticamente el código en módulos MVC, facilitando la escalabilidad.
- **ORM integrado:** Permite gestionar bases de datos sin escribir SQL directamente.
- **Seguridad incorporada:** Protege contra ataques comunes como **CSRF, XSS y SQL Injection**.

Ejemplo de estructura MVC en Django:

```
from django.http import HttpResponse
from django.shortcuts import render

def home(request):
    return HttpResponse("¡Bienvenido a Django!")
```

Cuándo usar Django:

- Para **proyectos grandes y escalables** con múltiples funcionalidades.
- Aplicaciones empresariales con **gestión de usuarios y bases de datos complejas**.
- Proyectos que requieren **seguridad y herramientas integradas**.

3. Core Patterns: Patrones Avanzados en MVC

3.1. Introducción a los Core Patterns en Arquitecturas MVC

Los **Core Patterns** son patrones de diseño avanzados que se utilizan en arquitecturas **Model-View-Controller (MVC)** para mejorar la organización, reutilización y mantenimiento del código. Estos patrones complementan la estructura MVC, optimizando la gestión de datos y la interacción entre componentes. Los **Core Patterns** refuerzan el diseño **MVC** al mejorar la organización del acceso a datos. Su integración con el modelo garantiza que la lógica de negocio se mantenga independiente de la capa de persistencia, promoviendo aplicaciones **modulares, escalables y reutilizables**. Su uso es clave en proyectos que buscan **mantenimiento eficiente y adaptación a largo plazo**.

¿Qué son los Core Patterns y por qué son útiles?

Los **Core Patterns** son patrones que proporcionan soluciones estructuradas a problemas recurrentes en el desarrollo de software. En el contexto de MVC, ayudan a **refinar la separación de responsabilidades**, especialmente en la gestión del acceso a datos y la comunicación entre el modelo y el resto de la aplicación.

Beneficios de los Core Patterns en MVC:

- **Modularidad:** Permiten que cada componente de la aplicación se mantenga independiente y fácil de actualizar.
- **Mantenimiento y escalabilidad:** Facilitan la adaptación a nuevos requisitos sin afectar toda la estructura del código.
- **Reutilización:** Los patrones pueden aplicarse en diferentes proyectos, reduciendo la redundancia en la programación.
- **Flexibilidad en el acceso a datos:** Se pueden cambiar o extender las fuentes de datos sin modificar otras capas de la aplicación.

Algunos de los Core Patterns más utilizados en MVC incluyen:

- **DAO (Data Access Object):** Abstrae la lógica de acceso a datos y evita que la lógica de negocio interactúe directamente con la base de datos.
- **DTO (Data Transfer Object):** Facilita la transferencia de datos entre capas sin exponer la estructura interna del modelo.
- **Repository Pattern:** Gestiona las operaciones de datos de manera centralizada, proporcionando una capa intermedia entre la lógica de negocio y la base de datos.

Relación entre MVC y Patrones de Acceso a Datos: En la arquitectura **MVC**, el modelo (Model) es responsable de la gestión de datos y la lógica de negocio. Sin embargo, a medida que las aplicaciones crecen, el acceso a los datos se vuelve más complejo y requiere un enfoque estructurado. Aquí es donde entran en juego los **patrones de acceso a datos** como DAO, DTO y Repository.

Ejemplo de cómo se integran estos patrones con MVC:

Modelo (Model) en MVC:

- Define la estructura de los datos y la lógica de negocio.
- Implementa validaciones y reglas de negocio.

DAO (Data Access Object):

- Separa la lógica de acceso a datos del modelo.
- Encapsula las consultas a la base de datos y evita dependencias directas con el código de la lógica de negocio.

DTO (Data Transfer Object):

- Transporta datos entre la base de datos y la aplicación sin exponer directamente la estructura del modelo.
- Reduce el acoplamiento entre capas y mejora la seguridad.

Repository Pattern:

- Actúa como una interfaz de alto nivel para gestionar entidades en la base de datos.
- Permite cambiar la fuente de datos sin afectar el código de la aplicación.

Ejemplo de Implementación en Python con Flask

```
class ProductoDTO:
    def __init__(self, id, nombre, precio):
        self.id = id
        self.nombre = nombre
        self.precio = precio

class ProductoDAO:
    def obtener_producto(self, id):
        # Simulación de acceso a base de datos
        data = {"id": id, "nombre": "Laptop", "precio": 1200}
        return ProductoDTO(data["id"], data["nombre"], data["precio"])

class ProductoRepository:
    def __init__(self):
        self.dao = ProductoDAO()

    def obtener_producto_con_formato(self, id):
        producto = self.dao.obtener_producto(id)
        return f"Producto: {producto.nombre}, Precio: {producto.precio}$"

# Uso del Repository en un controlador
repository = ProductoRepository()
print(repository.obtener_producto_con_formato(1))
```


3.2. DAO (Data Access Object) en Python

El **patrón Data Access Object (DAO)** es un patrón de diseño que abstrae y encapsula la lógica de acceso a bases de datos, proporcionando una interfaz uniforme para interactuar con los datos sin exponer los detalles de implementación. Este patrón es ampliamente utilizado en aplicaciones **MVC** para mantener la separación de responsabilidades y facilitar el mantenimiento del código. El **patrón DAO** es esencial en aplicaciones MVC porque permite manejar el acceso a la base de datos de manera estructurada y reutilizable. Su implementación en **Flask y Django** ayuda a mejorar la organización del código, facilitar el mantenimiento y aumentar la seguridad de la aplicación.

Propósito y Ventajas del DAO: El principal propósito del **DAO** es proporcionar una capa intermedia entre la lógica de negocio y la base de datos. En lugar de que las consultas SQL se escriban directamente en los controladores o modelos, DAO maneja todas las interacciones con la base de datos, asegurando un código más modular y reutilizable.

Ventajas del DAO:

- **Separación de responsabilidades:** Permite que la lógica de negocio se mantenga independiente del acceso a datos, facilitando el mantenimiento.
- **Facilita la migración de bases de datos:** Si la aplicación cambia de motor de base de datos (por ejemplo, de SQLite a PostgreSQL), solo es necesario modificar la capa DAO.
- **Reutilización de código:** Se pueden reutilizar métodos de acceso a datos en diferentes partes de la aplicación.
- **Mayor seguridad:** DAO permite centralizar el uso de consultas parametrizadas, reduciendo el riesgo de ataques **SQL Injection**.

Implementación del DAO en Flask/Django para Gestionar Consultas a la Base de Datos:

Ejemplo de DAO en Flask con SQLite

```
import sqlite3

class ProductoDAO:
    def __init__(self, db_path="tienda.db"):
        self.db_path = db_path

    def conectar(self):
        return sqlite3.connect(self.db_path)

    def obtener_productos(self):
        conexion = self.conectar()
        cursor = conexion.cursor()
        cursor.execute("SELECT id, nombre, precio FROM productos")
        productos = cursor.fetchall()
        conexion.close()
        return productos

    def agregar_producto(self, nombre, precio):
        conexion = self.conectar()
        cursor = conexion.cursor()
        cursor.execute("INSERT INTO productos (nombre, precio) VALUES (?, ?)", (nombre, precio))
        conexion.commit()
        conexion.close()
```

Uso del DAO en un controlador Flask:

```
from flask import Flask, render_template
from producto_dao import ProductoDAO

app = Flask(__name__)
producto_dao = ProductoDAO()

@app.route('/productos')
def mostrar_productos():
    productos = producto_dao.obtener_productos()
    return render_template("productos.html", productos=productos)

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

Aquí, ProductoDAO maneja el acceso a la base de datos, y el controlador Flask llama a su método obtener_productos() para pasar la información a la vista.

Ejemplo de DAO en Django con el ORM:

En **Django**, el ORM facilita la implementación del patrón DAO al encapsular la lógica de acceso a la base de datos dentro de modelos.

```
from django.db import models

class Producto(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.FloatField()

class ProductoDAO:
    @staticmethod
    def obtener_productos():
        return Producto.objects.all()

    @staticmethod
    def agregar_producto(nombre, precio):
        producto = Producto(nombre=nombre, precio=precio)
        producto.save()
```

Uso del DAO en una vista Django:

```
from django.shortcuts import render
from .dao import ProductoDAO

def lista_productos(request):
    productos = ProductoDAO.obtener_productos()
    return render(request, "productos.html", {"productos": productos})
```

Aquí, ProductoDAO actúa como intermediario entre la base de datos y la vista, siguiendo el principio de **separación de responsabilidades**.

3.3. DTO (Data Transfer Object) en Python

El patrón **DTO (Data Transfer Object)** es un patrón de diseño utilizado para transferir datos entre diferentes capas de una aplicación sin exponer directamente los modelos de base de datos. Su uso en una arquitectura **MVC (Model-View-Controller)** permite mejorar la seguridad, el mantenimiento y la independencia entre componentes. El uso de **DTO** en una arquitectura MVC en Python mejora la **seguridad, modularidad y mantenimiento** del sistema. Al evitar el paso directo de objetos del modelo, se reduce el **acoplamiento** y se controla mejor la exposición de datos. Este patrón es esencial en aplicaciones escalables donde los datos deben manipularse de manera controlada antes de enviarlos a la vista o al cliente.

¿Por qué usar DTO en lugar de pasar objetos de base de datos directamente?

En muchas aplicaciones, especialmente en **MVC con frameworks como Flask o Django**, los datos deben transferirse entre la capa de acceso a datos (Modelo), la capa de control (Controlador) y la capa de presentación (Vista). Sin DTO, los objetos del modelo de base de datos suelen pasarse directamente a la vista o al controlador, lo que puede generar problemas como:

- **Alta dependencia del modelo:** Si la estructura de la base de datos cambia, todas las partes del sistema que usan estos objetos también deben modificarse.
- **Exposición innecesaria de datos:** Puede exponer datos sensibles que no deberían enviarse a la vista o al cliente (por ejemplo, contraseñas o tokens).
- **Dificultad en la validación y transformación:** Los DTO permiten estructurar los datos en un formato adecuado antes de enviarlos a la vista o a una API.
- **Facilita la serialización:** DTO permite transformar los datos en formatos como JSON sin depender directamente del modelo de base de datos.

Creación y Uso de DTO en una Aplicación MVC en Python:

1. Definición del Modelo en Django

Supongamos que tenemos un modelo Usuario en Django que representa los datos de un usuario en la base de datos:

```
from django.db import models

class Usuario(models.Model):
    id = models.AutoField(primary_key=True)
    nombre = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    password = models.CharField(max_length=255)  # No debería exponerse
    fecha_registro = models.DateTimeField(auto_now_add=True)
```

Si pasamos directamente este modelo a la vista, el usuario podría ver información que no debería estar expuesta, como la contraseña.

2. Creación de un DTO para Usuario

Para evitar este problema, creamos un **DTO** que solo contenga los datos relevantes:

```
class UsuarioDTO:
    def __init__(self, id, nombre, email, fecha_registro):
        self.id = id
        self.nombre = nombre
        self.email = email
        self.fecha_registro = fecha_registro
```

Aquí, **excluimos el campo password**, lo que evita exponer información sensible al usuario.

3. Implementación del DAO con DTO

Creamos un **DAO (Data Access Object)** para obtener datos de la base de datos y transformarlos en un objeto DTO:

```
class UsuarioDAO:
    @staticmethod
    def obtener_usuarios():
        usuarios = Usuario.objects.all()
        return [UsuarioDTO(u.id, u.nombre, u.email, u.fecha_registro) for u in usuarios]
```

Ahora, en lugar de pasar objetos Usuario directamente, pasamos **objetos DTO**, asegurando que solo se transfieran los datos necesarios.

4. Uso del DTO en un Controlador Django

```
from django.http import JsonResponse
from .dao import UsuarioDAO

def lista_usuarios(request):
    usuarios_dto = UsuarioDAO.obtener_usuarios()
    usuarios_serializados = [{"id": u.id, "nombre": u.nombre, "email": u.email, "fecha_registro": u.fecha_registro}
for u in usuarios_dto]
    return JsonResponse({"usuarios": usuarios_serializados})
```

Aquí, convertimos los **DTO** en un formato JSON adecuado para ser enviado como respuesta en una API REST.

3.4. Repository Pattern en MVC

El **Repository Pattern** es un patrón de diseño que actúa como una **capa intermedia entre la lógica de negocio y la capa de acceso a datos**, proporcionando una interfaz común para interactuar con los datos. Se usa para desacoplar el código de la base de datos, facilitando la reutilización, la prueba y la migración a diferentes tecnologías de persistencia. El **Repository Pattern** proporciona una capa de abstracción que mejora la organización del código en aplicaciones MVC. Su combinación con **DAO** permite un acceso a datos estructurado, modular y fácil de mantener. Es una excelente práctica en proyectos grandes donde la escalabilidad y el mantenimiento son clave.

Definición y Beneficios del Patrón Repository: El **Repository Pattern** permite gestionar el acceso a los datos sin que la lógica de negocio tenga que interactuar directamente con la base de datos. En lugar de realizar consultas SQL en los controladores o modelos, estas se encapsulan en la capa de **Repositorio**, lo que hace que la aplicación sea más modular y fácil de mantener.

Beneficios del patrón Repository:

- **Desacoplamiento del código:** La lógica de negocio no depende directamente del acceso a datos, lo que facilita cambios en la base de datos sin afectar el resto del sistema.
- **Reutilización:** Se pueden reutilizar las funciones de acceso a datos en diferentes partes de la aplicación.
- **Facilita la prueba de unidades:** Al separar la base de datos de la lógica de negocio, se pueden hacer pruebas unitarias con datos simulados.
- **Compatibilidad con diferentes fuentes de datos:** Permite cambiar entre bases de datos SQL y NoSQL sin modificar la lógica de negocio.

Diferencias con DAO y Cuándo Usarlos Juntos:

El **DAO (Data Access Object)** y el **Repository Pattern** tienen propósitos similares, pero con diferencias clave:

Característica	DAO	Repository Pattern
Propósito	Encapsular la lógica de acceso a datos y consultas SQL.	Proporcionar una abstracción de la base de datos para la lógica de negocio.
Nivel de abstracción	Bajo (interactúa directamente con la base de datos).	Alto (actúa como intermediario entre DAO y lógica de negocio).
Relación con MVC	Pertenece a la capa de acceso a datos.	Se encuentra entre el modelo y la lógica de negocio.
Uso en la aplicación	Se usa para realizar consultas SQL directas.	Se usa para proporcionar métodos de alto nivel para acceder a los datos.

¿Cuándo usarlos juntos?: El **DAO** puede encargarse de ejecutar las consultas SQL y el **Repository** de gestionar la lógica de acceso a los datos. Esto permite un diseño aún más modular y reutilizable.

Implementación Práctica en Python con Flask y Django:

Ejemplo en Flask

```
# Modelo de Producto
class Producto:
    def __init__(self, id, nombre, precio):
        self.id = id
        self.nombre = nombre
        self.precio = precio

# DAO para acceso a base de datos
class ProductoDAO:
    def obtener_producto(self, id):
        # Simulación de base de datos
        data = {"id": id, "nombre": "Laptop", "precio": 1200}
        return Producto(data["id"], data["nombre"], data["precio"])

# Repositorio que usa DAO
class ProductoRepository:
    def __init__(self):
        self.dao = ProductoDAO()

    def obtener_producto_con_formato(self, id):
        producto = self.dao.obtener_producto(id)
        return f"Producto: {producto.nombre}, Precio: {producto.precio}$"

# Uso en un controlador Flask
repository = ProductoRepository()
print(repository.obtener_producto_con_formato(1))
```

Ejemplo en Django

```
from django.db import models

# Modelo de Producto
class Producto(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.FloatField()

# Repositorio
class ProductoRepository:
    @staticmethod
    def obtener_productos():
        return Producto.objects.all()

    @staticmethod
    def agregar_producto(nombre, precio):
        producto = Producto(nombre=nombre, precio=precio)
        producto.save()
```

Uso en una vista Django:

```
from django.shortcuts import render
from .repository import ProductoRepository

def lista_productos(request):
    productos = ProductoRepository.obtener_productos()
    return render(request, "productos.html", {"productos": productos})
```

Bibliografía y lecturas recomendadas:

- **Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994).** *Patrones de Diseño: Elementos Reutilizables en la Programación Orientada a Objetos.* Addison-Wesley.
- **Sánchez, M. (2021).** *Desarrollo web con Flask y Django.* Ediciones Ra-Ma.
- **Pressman, R. S., & Maxim, B. R. (2015).** *Ingeniería del Software: Un Enfoque Práctico.* McGraw-Hill.
- **Gutiérrez, J. (2019).** *Patrones de Diseño en Python.* Alfaomega.
- **Documentación Oficial de Flask:** URL: https://flask.palletsprojects.com/en/2.3.x/
- **Django Project – Documentación Oficial:** URL: https://docs.djangoproject.com/es/4.2/
- **Refactoring.Guru – Patrones de Diseño:** URL: https://refactoring.guru/es/design-patterns

Actividades prácticas

Ejercicio 13. Implementación de MVC en una Aplicación Web con Flask

Un equipo de desarrollo está construyendo un sistema de gestión de tareas basado en el patrón Model-View-Controller (MVC). Se te ha asignado la tarea de diseñar la estructura del sistema en Flask, asegurando la separación de responsabilidades.

Objetivo:
Implementar una aplicación web sencilla que permita:

1. Agregar nuevas tareas.
2. Ver la lista de tareas registradas.
3. Marcar tareas como completadas.

1. Tareas a realizar:

Definir el modelo (Task). Debe contener un ID, un nombre de tarea y un estado (pendiente o completado).
Implementar el controlador. Debe gestionar las solicitudes de los usuarios y comunicarse con el modelo.
Crear la vista. Debe mostrar la lista de tareas y permitir agregar nuevas.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos

Ejercicio 14. Análisis y Diseño de una Aplicación Web con el Patrón MVC

Una empresa de gestión de eventos desea desarrollar una aplicación web que permita a los usuarios registrarse en eventos, ver la lista de eventos disponibles y gestionar sus inscripciones.

El equipo de desarrollo ha decidido implementar el patrón Model-View-Controller (MVC) para garantizar una estructura modular y escalable. Como analista del proyecto, tu tarea es diseñar la arquitectura de la aplicación sin necesidad de programar código, identificando cómo se distribuirán las responsabilidades en cada componente de MVC.

1. Definir el Modelo (Model). ¿Qué entidades y atributos deben manejarse en la aplicación?
   Diseñar la Vista (View). ¿Qué pantallas y elementos visuales necesita el usuario para interactuar con la aplicación?
   Especificar el Controlador (Controller). ¿Qué lógica de negocio debe implementarse para conectar la vista con el modelo?
   Explicar cómo la implementación de MVC mejora la organización y mantenimiento del sistema.

Procesando respuesta, no cierres el navegador, este proceso podría tardar unos segundos