🗄️ Conexión con MySQL

Conectarse a la Base de Datos y Recuperar Datos

Python Full Stack - Clase 24

¿Qué aprenderemos hoy?

- 🔌 Conectarnos a MySQL desde Flask usando PyMySQL
- 🏗️ Crear la clase MySQLConnection para manejar conexiones
- 📦 Instalar y configurar PyMySQL en nuestro proyecto
- 🎯 Crear modelos con OOP basados en nuestras tablas
- 📊 Recuperar datos de la base de datos (SELECT)
- 🖥️ Visualizar datos en nuestras plantillas HTML
- 🔐 Usar sentencias preparadas para consultas con variables
- ⚠️ Manejar errores comunes de conexión

🔌 Conectarse a MySQL

¿Por Qué Necesitamos Conectarnos?

Hasta ahora hemos trabajado con:

- ✅ Formularios que capturan datos
- ✅ Sesiones que almacenan información temporalmente
- ✅ Plantillas que muestran contenido

Pero falta algo importante: 💾

¿Dónde guardamos los datos de forma permanente?

PyMySQL: El Puente entre Flask y MySQL

PyMySQL es una librería que nos permite:

- 🔌 Conectarnos al servidor MySQL
- 📝 Ejecutar consultas SQL
- 📊 Recibir resultados de las consultas
- 🔒 Protegernos contra SQL injection

Es como un traductor entre Python y MySQL.

Instalando PyMySQL

Para instalar PyMySQL en nuestro proyecto:

```
pipenv install flask pymysql
```

Este comando hace **dos cosas**:

1. ✅ Crea/actualiza el entorno virtual
2. ✅ Instala Flask y PyMySQL

⚠️ **Importante:** Siempre ejecuta esto desde la carpeta de tu proyecto.

Estructura del Proyecto

Antes de empezar, organicemos nuestro proyecto:

```
proyecto_tienda/
   ├── flask_app/
   │    ├── __init__.py
   │    ├── config/
   │    │    └── mysqlconnection.py    ← Conexión a BD
   │    ├── models/
   │    │    ├── usuario.py            ← Modelo Usuario
   │    │    ├── producto.py           ← Modelo Producto
   │    │    └── categoria.py          ← Modelo Categoria
   │    ├── controllers/
   │    │    └── productos_controller.py
   │    ├── templates/
   │    │    └── dashboard.html
   │    └── static/
   └── server.py
```

Creando la Conexión

La Clase MySQLConnection

Necesitamos una clase que maneje la conexión con MySQL:

```
import pymysql.cursors

class MySQLConnection:
    def __init__(self, db):
        # Establece la conexión con la base de datos
        connection = pymysql.connect(
            host='localhost',
            user='root',           # Tu usuario de MySQL
            password='',           # Tu contraseña de MySQL (cambia esto)
            db=db,                 # Nombre de la base de datos
            charset='utf8mb4',
            cursorclass=pymysql.cursors.DictCursor,
            autocommit=True
        )
        self.connection = connection
```

Desglosando los Parámetros

`host='localhost'`

- Dirección del servidor MySQL (local en tu computadora)

`user='root'` y `password=''`

- Credenciales para conectarte a MySQL
- ⚠️ **Cambia estos valores** por los tuyos (especialmente la contraseña)

`db=db`

- Nombre de la base de datos que queremos usar

`charset='utf8mb4'`

- Permite caracteres especiales (tildes, emojis, etc.)

Más Parámetros Importantes

`cursorclass=pymysql.cursors.DictCursor`

- Los resultados vienen como **diccionarios** en lugar de tuplas
- Más fácil de trabajar: `resultado['nombre']` en vez de `resultado[0]`

`autocommit=True`

- Los cambios se guardan automáticamente
- No necesitamos hacer `commit()` manualmente

El Método query_db

```
def query_db(self, query, data=None):
    with self.connection.cursor() as cursor:
        try:
            query = cursor.mogrify(query, data)
            print("Running Query:", query)

            executable = cursor.execute(query, data)

            # Manejar diferentes tipos de consultas
            if query.lower().find("insert") >= 0:
                self.connection.commit()
                return cursor.lastrowid
            elif query.lower().find("select") >= 0:
                result = cursor.fetchall()
                return result
            else:
                self.connection.commit()
        except Exception as e:
            print("Something went wrong", e)
            return False
        finally:
            self.connection.close()
```

¿Qué Hace query_db?

Entrada:

- `query` : La consulta SQL que queremos ejecutar
- `data` : Diccionario con valores para sentencias preparadas (opcional)

Salida según el tipo de consulta:

- **SELECT**: Lista de diccionarios con los resultados
- **INSERT**: El ID del nuevo registro creado
- **UPDATE/DELETE**: Nada (solo confirma que se ejecutó)
- **Error**: `False` si algo sale mal

La Función Helper

Para facilitar el uso, creamos una función:

```
def connectToMySQL(db):
    return MySQLConnection(db)
```

Uso:

```
# Conecta a la base de datos 'tienda'
conexion = connectToMySQL('tienda')
resultados = conexion.query_db("SELECT * FROM usuarios;")
```

¿Por qué una función? Hace el código más limpio y fácil de leer.

Archivo Completo: mysqlconnection.py

```
import pymysql.cursors

class MySQLConnection:
    def __init__(self, db):
        connection = pymysql.connect(
            host='localhost',
            user='root',        # Cambia por tu usuario
            password='',        # Cambia por tu contraseña
            db=db,
            charset='utf8mb4',
            cursorclass=pymysql.cursors.DictCursor,
            autocommit=True
        )
        self.connection = connection

    def query_db(self, query, data=None):
        with self.connection.cursor() as cursor:
            try:
                query = cursor.mogrify(query, data)
                print("Running Query:", query)

                executable = cursor.execute(query, data)

                if query.lower().find("insert") >= 0:
                    self.connection.commit()
                    return cursor.lastrowid
                elif query.lower().find("select") >= 0:
                    result = cursor.fetchall()
                    return result
                else:
                    self.connection.commit()
            except Exception as e:
                print("Something went wrong", e)
                return False
            finally:
                self.connection.close()

def connectToMySQL(db):
    return MySQLConnection(db)
```

⚠️ Errores Comunes de Conexión

Error 1: Credenciales Incorrectas

Error:

```
Access denied for user 'root'@'localhost'
```

Causa: Usuario o contraseña incorrectos

Solución:

- Verifica tus credenciales de MySQL
- Asegúrate de usar las mismas que en MySQL Workbench

Error 2: Base de Datos No Existe

Error:

```
Unknown database 'tienda'
```

Causa: La base de datos no ha sido creada

Solución:

1. Abre MySQL Workbench
2. Ejecuta el script SQL `00-e-commerce.sql` para crear la base de datos `tienda`
3. O crea manualmente la base de datos `tienda` y ejecuta el script

Error 3: Servidor No Está Corriendo

Error:

```
Can't connect to MySQL server on 'localhost'
```

Causa: El servidor MySQL no está ejecutándose

Solución:

- En Windows: Inicia MySQL desde el Administrador de Servicios
- En Mac: `sudo /usr/local/mysql/support-files/mysql.server start`
- Verifica que MySQL esté corriendo antes de ejecutar Flask

Error 4: Puerto Incorrecto

Error:

```
Can't connect to MySQL server on 'localhost' (port 3306)
```

Causa: MySQL está usando un puerto diferente

Solución:

- Verifica el puerto en MySQL Workbench
- Si es diferente, agrega `port=3307` (o el que uses) en la conexión

💪 Ejercicio: Probar la Conexión

¡Hora de practicar!

Crea un archivo `test_conexion.py`:

1. Importa `connectToMySQL` desde `mysqlconnection.py`
2. Conéctate a la base de datos `tienda`
3. Ejecuta la consulta: `SELECT * FROM usuarios;`
4. Imprime los resultados

Pistas:

- Usa `print()` para ver qué devuelve la consulta
- Verifica que tu base de datos tenga datos de prueba

⏱️ **Tiempo:** 10 minutos

🎯 Creando Modelos con OOP

¿Qué es un Modelo?

Un **modelo** es una clase de Python que representa una tabla de la base de datos.

Ejemplo:

- Tabla `usuarios` → Clase `Usuario`
- Tabla `productos` → Clase `Producto`
- Tabla `categorias` → Clase `Categoria`

Ventajas:

- ✅ Código organizado y reutilizable
- ✅ Fácil de mantener
- ✅ Sigue el patrón MVC

Estructura de un Modelo

Un modelo tiene dos partes principales:

1. `__init__`: Crea un objeto a partir de un diccionario
2. **Métodos de clase**: Consultas a la base de datos

Ejemplo básico:

```
class Usuario:
    def __init__(self, data):
        self.id = data['id']
        self.nombre = data['nombre']
        # ... más atributos

    @classmethod
    def get_all(cls):
        # Consulta que devuelve todos los usuarios
        pass
```

Creando el Modelo Usuario

Basado en la tabla `usuarios` del proyecto:

```
from flask_app.config.mysqlconnection import connectToMySQL

class Usuario:
    def __init__(self, data):
        self.id = data['id']
        self.nombre = data['nombre']
        self.apellido = data['apellido']
        self.email = data['email']
        self.created_at = data['created_at']
        self.updated_at = data['updated_at']
```

Cada atributo corresponde a una columna de la tabla.

Método get_all: Obtener Todos los Registros

```python
@classmethod
def get_all(cls):
    query = "SELECT * FROM usuarios;"
    resultados = connectToMySQL('tienda').query_db(query)

    usuarios = []
    for usuario in resultados:
        usuarios.append(cls(usuario))

    return usuarios
```

¿Qué hace?

1. Ejecuta `SELECT * FROM usuarios`
2. Recibe una lista de diccionarios
3. Convierte cada diccionario en un objeto `Usuario`
4. Retorna una lista de objetos `Usuario`

¿Por Qué Usar @classmethod?

`@classmethod` permite llamar al método sin crear una instancia:

```
# ✅ Correcto - Método de clase
usuarios = Usuario.get_all()

# ❌ Incorrecto - No necesitamos crear un objeto primero
usuario = Usuario(...)
usuarios = usuario.get_all()
```

Ventaja: Más limpio y semánticamente correcto.

Método get_one: Obtener un Solo Registro

```
@classmethod
def get_one(cls, data):
    query = "SELECT * FROM usuarios WHERE id = %(id)s;"
    resultado = connectToMySQL('tienda').query_db(query, data)

    if resultado:
        return cls(resultado[0])
    return False
```

Uso:

```
data = {'id': 1}
usuario = Usuario.get_one(data)
print(usuario.nombre)  # Imprime el nombre del usuario
```

Creando el Modelo Producto

```python
from flask_app.config.mysqlconnection import connectToMySQL

class Producto:
    def __init__(self, data):
        self.id = data['id']
        self.nombre = data['nombre']
        self.precio = data['precio']
        self.descripcion = data['descripcion']
        self.categoria_id = data['categoria_id']
        self.created_at = data['created_at']
        self.updated_at = data['updated_at']

    @classmethod
    def get_all(cls):
        query = "SELECT * FROM productos;"
        resultados = connectToMySQL('tienda').query_db(query)

        productos = []
        for producto in resultados:
            productos.append(cls(producto))

        return productos
```

💪 Ejercicio: Crear Modelo Producto Completo

¡Hora de practicar!

Crea el archivo `flask_app/models/producto.py`:

1. Crea la clase `Producto` con `__init__`
2. Agrega el método `get_all()` que devuelva todos los productos
3. Agrega el método `get_one(data)` que devuelva un producto por ID
4. Prueba ambos métodos en `server.py`

Pistas:

- Usa el modelo `Usuario` como referencia
- Recuerda importar `connectToMySQL`
- El nombre de la base de datos es `tienda`
- Los atributos del producto son: id, nombre, precio, descripcion, categoria_id, created_at, updated_at

⏱️ **Tiempo:** 15 minutos

Recuperar y Visualizar Datos

El Flujo Completo

```
🗄️ Base de Datos MySQL
    ↓ (SELECT query)
🐍 Modelo (Producto.get_all())
    ↓ (lista de objetos)
🎮 Controlador (server.py)
    ↓ (render_template)
📄 Plantilla HTML
    ↓ (Jinja2)
🌐 Navegador
```

Cada paso tiene un propósito específico.

En el Controlador (server.py)

```python
from flask import Flask, render_template
from flask_app.models.producto import Producto

app = Flask(__name__)

@app.route("/productos")
def mostrar_productos():
    productos = Producto.get_all()
    print(productos)  # Ver en la terminal
    return render_template("dashboard.html", todos_productos=productos)
```

¿Qué pasa aquí?

1. Llamamos a `Producto.get_all()` → Obtiene todos los productos
2. Pasamos la lista a la plantilla como `todos_productos`
3. La plantilla puede iterar sobre esta lista

En la Plantilla (dashboard.html)

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Productos</title>
</head>
<body>
    <h1>Lista de Productos</h1>

    {% for producto in todos_productos %}
    <div>
        <h3>{{ producto.nombre }}</h3>
        <p><strong>Precio:</strong> ${{ producto.precio }}</p>
        <p><strong>Descripción:</strong> {{ producto.descripcion }}</p>
        <p><strong>Categoría ID:</strong> {{ producto.categoria_id }}</p>
    </div>
    <hr>
    {% endfor %}
</body>
</html>
```

¿Qué Devuelve get_all()?

En Python (server.py):

```
productos = Producto.get_all()
# productos es una LISTA de objetos Producto
# [
#   <Producto: Laptop HP>,
#   <Producto: Smartphone Samsung>,
#   <Producto: Camiseta Algodón>
# ]
```

En la plantilla:

- `todos_productos` es la misma lista
- Podemos iterar con `{% for %}`
- Accedemos a atributos con `{{ producto.nombre }}`

Mostrando Datos en una Tabla

```
<table>
    <thead>
        <tr>
            <th>Nombre</th>
            <th>Precio</th>
            <th>Descripción</th>
        </tr>
    </thead>
    <tbody>
        {% for producto in todos_productos %}
        <tr>
            <td>{{ producto.nombre }}</td>
            <td>${{ producto.precio }}</td>
            <td>{{ producto.descripcion }}</td>
        </tr>
        {% endfor %}
    </tbody>
</table>
```

Resultado: Una tabla HTML con todos los productos.

Mostrando un Solo Producto

En server.py:

```
@app.route("/productos/<int:id>")
def ver_producto(id):
    data = {'id': id}
    producto = Producto.get_one(data)

    if producto:
        return render_template("ver_producto.html", producto=producto)
    else:
        return "Producto no encontrado", 404
```

En ver_producto.html:

```
<h1>{{ producto.nombre }}</h1>
<p><strong>Precio:</strong> ${{ producto.precio }}</p>
<p><strong>Descripción:</strong> {{ producto.descripcion }}</p>
<p><strong>Categoría ID:</strong> {{ producto.categoria_id }}</p>
```

💪 Ejercicio: Dashboard de Productos

¡Hora de practicar!

Crea una ruta `/productos` que muestre todos los productos:

1. Usa el modelo `Producto` para obtener todos los productos
2. Crea una plantilla `dashboard.html` que muestre:
    - Nombre del producto
    - Precio (formateado como moneda)
    - Descripción
    - Categoría ID
3. Muestra los productos en una tabla HTML con estilo

Pistas:

- Usa `{% for %}` para iterar
- Agrega CSS básico para que se vea bien
- Verifica que la base de datos tenga datos (ejecuta el script SQL primero)
- Para formatear el precio, usa: `${{ producto.precio }}`

⏱️ **Tiempo:** 20 minutos

🔐 Consultas con Datos Variables

El Problema: Consultas Estáticas

Hasta ahora hemos usado consultas fijas:

```
query = "SELECT * FROM productos;"
query = "SELECT * FROM usuarios WHERE id = 1;"
```

¿Qué pasa si queremos buscar por diferentes IDs? 🤔

Necesitamos hacer la consulta **dinámica**.

La Solución: Sentencias Preparadas

Las **sentencias preparadas** nos permiten usar variables de forma segura:

```
query = "SELECT * FROM usuarios WHERE id = %(id)s;"
data = {'id': 1}
resultado = connectToMySQL('tienda').query_db(query, data)
```

Ventajas:

- ✅ Seguro contra SQL injection
- ✅ Reutilizable con diferentes valores
- ✅ Más legible

Sintaxis de Sentencias Preparadas

Formato:

```
query = "SELECT * FROM tabla WHERE columna = %(clave)s;"
data = {'clave': valor}
```

Reglas:

- Usa `%(clave)s` en el query (con `s` al final)
- La `clave` debe coincidir con la clave del diccionario
- Siempre pasa `data` como segundo parámetro

Ejemplo 1: Buscar por ID

```
@classmethod
def get_one(cls, data):
    query = "SELECT * FROM usuarios WHERE id = %(id)s;"
    resultado = connectToMySQL('tienda').query_db(query, data)

    if resultado:
        return cls(resultado[0])
    return False

# Uso:
usuario = Usuario.get_one({'id': 1})
```

Ejemplo 2: Buscar por Email

```
@classmethod
def get_by_email(cls, data):
    query = "SELECT * FROM usuarios WHERE email = %(email)s;"
    resultado = connectToMySQL('tienda').query_db(query, data)

    if resultado:
        return cls(resultado[0])
    return False

# Uso:
usuario = Usuario.get_by_email({'email': 'juan.perez@email.com'})
```

Ejemplo 3: Múltiples Condiciones

```python
@classmethod
def get_productos_por_precio(cls, data):
    query = """
        SELECT * FROM productos
        WHERE precio >= %(precio_minimo)s
        AND precio <= %(precio_maximo)s;
    """
    resultados = connectToMySQL('tienda').query_db(query, data)

    productos = []
    for producto in resultados:
        productos.append(cls(producto))

    return productos

# Uso:
data = {
    'precio_minimo': 50.00,
    'precio_maximo': 500.00
}
productos = Producto.get_productos_por_precio(data)
```

Ejemplo 4: Buscar Productos por Categoría

```
@classmethod
def get_by_categoria(cls, data):
    query = "SELECT * FROM productos WHERE categoria_id = %(categoria_id)s;"
    resultados = connectToMySQL('tienda').query_db(query, data)

    productos = []
    for producto in resultados:
        productos.append(cls(producto))

    return productos

# Uso:
productos_electronica = Producto.get_by_categoria({'categoria_id': 1})
```

⚠️ Importante: SQL Injection

❌ NUNCA hagas esto:

```
# PELIGROSO - Vulnerable a SQL injection
query = f"SELECT * FROM usuarios WHERE email = '{email}';"
```

✅ SIEMPRE usa sentencias preparadas:

```
# SEGURO - Protegido contra SQL injection
query = "SELECT * FROM usuarios WHERE email = %(email)s;"
data = {'email': email}
```

¿Por qué? Las sentencias preparadas sanitizan los datos automáticamente.

¡Hora de practicar!

Agrega estos métodos al modelo `Producto`:

1. `get_by_categoria(data)`: Busca productos por categoría
2. `get_por_precio(data)`: Busca productos entre un precio mínimo y máximo
3. `buscar_en_descripcion(data)`: Busca productos que contengan un texto en la descripción

En server.py, crea rutas que usen estos métodos:

- `/productos/categoria/<int:categoria_id>`
- `/productos/precio/<float:minimo>/<float:maximo>`
- `/productos/buscar/<texto>`

Pistas:

- Usa `LIKE` para búsquedas de texto: `WHERE descripcion LIKE %(texto)s`
- Para LIKE, usa: `'%' + texto + '%'` en el diccionario
- Para precios, usa condiciones `WHERE precio >= %(minimo)s AND precio <= %`

Resumen de Conceptos Clave

✅ **PyMySQL**: Librería para conectar Python con MySQL
✅ **MySQLConnection**: Clase que maneja la conexión y consultas
✅ **Modelos**: Clases Python que representan tablas de BD
✅ **Métodos de clase**: Consultas organizadas dentro del modelo
✅ **Sentencias preparadas**: Forma segura de usar variables en queries
✅ **Renderizar datos**: Pasar objetos del modelo a plantillas HTML

Lo Más Importante

🎯 La conexión se cierra después de cada consulta

- Cada vez que llamamos `query_db()`, se abre y cierra la conexión
- Por eso llamamos `connectToMySQL()` en cada método

🎯 SELECT devuelve lista de diccionarios

- `get_all()` → Lista de objetos
- `get_one()` → Un objeto o `False`

🎯 Siempre usa sentencias preparadas

- `%(clave)s` en el query
- Diccionario `data` con los valores
- Protección contra SQL injection

Recursos para Seguir Aprendiendo

📚 **Documentación oficial:**

- PyMySQL: https://pymysql.readthedocs.io/
- MySQL: https://dev.mysql.com/doc/

💡 **Próximos pasos:**

- Crear registros (INSERT)
- Actualizar registros (UPDATE)
- Eliminar registros (DELETE)
- Relaciones entre tablas (JOINs)

Próximos Pasos

En la siguiente clase veremos:

- ➕ **Crear registros** (INSERT) desde formularios
- ✏️ **Actualizar registros** (UPDATE) con formularios de edición
- 🗑️ **Eliminar registros** (DELETE) con confirmación
- 🔗 **JOINs** para obtener datos relacionados

Práctica para Casa

🏠 **Proyecto sugerido:**

Completa el dashboard de productos:

- Muestra todos los productos en una tabla
- Agrega un enlace para ver detalles de cada producto
- Crea una página de detalle que muestre toda la información
- Agrega filtros por categoría y precio

Objetivo: Practicar recuperación de datos y visualización en plantillas.

Consejos Finales

✅ **Siempre verifica tus credenciales** de MySQL antes de empezar
✅ **Usa** `print()` **para depurar** y ver qué devuelven las consultas
✅ **Organiza tus modelos** en la carpeta `models/`
✅ **Nombra tus métodos claramente**: `get_all()`, `get_one()`, `get_by_email()`
✅ **Prueba tus consultas** en MySQL Workbench antes de usarlas en código

¡La práctica hace al maestro! 💪

🎉 ¡Felicidades!

Ya sabes conectarte a MySQL y recuperar datos

Has aprendido los fundamentos de bases de datos en Flask

? Preguntas

¿Alguna duda sobre conexión a MySQL o recuperación de datos?

🙌 ¡Excelente Trabajo!

¡Nos vemos en la siguiente clase con INSERT, UPDATE y DELETE!

No olvides completar todos los ejercicios 📝