🔗 Relaciones Muchos a Muchos

Extendiendo nuestro proyecto de Tacos

Python Full Stack - Clase 29

¿Qué aprenderemos hoy?

- 🔍 Entender qué son las relaciones muchos a muchos
- 📊 Crear tablas intermedias para relacionar entidades
- 💻 Implementar modelos con relaciones muchos a muchos
- 🔄 Usar LEFT JOIN para obtener datos relacionados
- 🎯 Evitar importaciones circulares en Python
- 🎨 Mostrar relaciones en nuestras vistas HTML
- 🛠️ Extender un proyecto existente con nuevas relaciones

🔍 Repasando Tipos de Relaciones

Tipos de Relaciones que Conocemos

Hasta ahora hemos trabajado con:

Uno a Muchos (1:N)

- Un restaurante tiene muchos tacos
- Un taco pertenece a un restaurante
- Se usa una **foreign key** en la tabla "muchos"

Muchos a Muchos (N:N) ← ¡NUEVO!

- Un taco puede tener muchos complementos
- Un complemento puede estar en muchos tacos
- Se necesita una **tabla intermedia**

¿Qué es una Relación Muchos a Muchos?

Imagina que quieres agregar **complementos** a tus tacos:

- 🌮 Un taco puede tener: cebolla, cilantro, limón, rábanos
- 🥗 Un complemento (como "cebolla") puede estar en muchos tacos diferentes

¡Esto es una relación muchos a muchos!

```
Taco 1 → Cebolla, Cilantro, Limón
Taco 2 → Cebolla, Limón, Queso
Taco 3 → Cilantro, Limón, Rábanos
```

Diseñando la Base de Datos

Paso 1: Crear la Tabla de Complementos

Primero, creamos la tabla principal de complementos:

```sql
CREATE TABLE IF NOT EXISTS `complementos` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `nombre_complemento` VARCHAR(45) NULL,
  `created_at` DATETIME NULL DEFAULT CURRENT_TIMESTAMP,
  `updated_at` DATETIME NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE = InnoDB;
```

Atributos:

- `id` : Identificador único (clave primaria)
- `nombre_complemento` : Nombre del complemento (ej: "Cebolla", "Cilantro")
- `created_at` y `updated_at` : Timestamps automáticos

Paso 2: Crear la Tabla Intermedia

```
CREATE TABLE IF NOT EXISTS `complementos_en_tacos` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `taco_id` INT NOT NULL,
  `complemento_id` INT NOT NULL,
  `created_at` DATETIME NULL DEFAULT CURRENT_TIMESTAMP,
  `updated_at` DATETIME NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  INDEX `fk_complementos_en_tacos_tacos_idx` (`taco_id` ASC),
  INDEX `fk_complementos_en_tacos_complementos_idx` (`complemento_id` ASC),
  CONSTRAINT `fk_complementos_en_tacos_tacos`
    FOREIGN KEY (`taco_id`)
    REFERENCES `tacos` (`id`)
    ON DELETE CASCADE
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_complementos_en_tacos_complementos`
    FOREIGN KEY (`complemento_id`)
    REFERENCES `complementos` (`id`)
    ON DELETE CASCADE
    ON UPDATE NO ACTION
) ENGINE = InnoDB;
```

Entendiendo la Tabla Intermedia

Campos importantes:

- `taco_id` : Referencia al taco (foreign key)
- `complemento_id` : Referencia al complemento (foreign key)
- **Ambos juntos** forman la relación

Ejemplo de datos:

```
taco_id | complemento_id
--------|---------------
   1    |       1        ← Taco 1 tiene Complemento 1 (Cebolla)
   1    |       2        ← Taco 1 tiene Complemento 2 (Cilantro)
   2    |       1        ← Taco 2 tiene Complemento 1 (Cebolla)
```

¿Por qué una Tabla Intermedia?

Sin tabla intermedia (❌ IMPOSIBLE):

```
tacos
  id | complemento_id  ← ¿Cuál complemento? ¿Solo uno?
```

Con tabla intermedia (✅ CORRECTO):

```
complementos_en_tacos
  taco_id | complemento_id
    1     |       1
    1     |       2
    1     |       3
```

Permite **múltiples relaciones** entre las mismas entidades.

💻 Creando el Modelo Complemento

Creando la Clase Complemento

Creamos el archivo `flask_app/models/complemento.py`:

```
from flask_app.config.mysqlconnection import connectToMySQL
from flask_app.models import taco  # Importamos el módulo (no la clase)

class Complemento:
    def __init__(self, data):
        self.id = data['id']
        self.nombre_complemento = data['nombre_complemento']
        # Lista para almacenar los tacos relacionados
        self.en_tacos = []
        self.created_at = data['created_at']
        self.updated_at = data['updated_at']
```

Nota importante: Importamos el **módulo** `taco`, no la clase `Taco`. Esto evita importaciones circulares.

Métodos Básicos del Modelo Complemento

```python
@classmethod
def save(cls, datos):
    query = "INSERT INTO complementos (nombre_complemento) VALUES (%(nombre_complemento)s)"
    return connectToMySQL('esquema_tacos').query_db(query, datos)

@classmethod
def get_all(cls):
    query = "SELECT * FROM complementos;"
    complementos_en_bd = connectToMySQL('esquema_tacos').query_db(query)
    complementos = []
    for complemento in complementos_en_bd:
        complementos.append(cls(complemento))
    return complementos

@classmethod
def get_one(cls, datos):
    query = "SELECT * FROM complementos WHERE id = %(id)s;"
    complemento_en_db = connectToMySQL('esquema_tacos').query_db(query, datos)
    if complemento_en_db:
        return cls(complemento_en_db[0])
    return None
```

Método para Obtener Complemento con Tacos

Este es el método **clave** para relaciones muchos a muchos:

```python
@classmethod
def get_complemento_y_tacos(cls, datos):
    query = """
        SELECT * FROM complementos
        LEFT JOIN complementos_en_tacos ON complementos_en_tacos.complemento_id = complementos.id
        LEFT JOIN tacos ON complementos_en_tacos.taco_id = tacos.id
        WHERE complementos.id = %(id)s;
    """
    resultados = connectToMySQL('esquema_tacos').query_db(query, datos)

    if not resultados or not resultados[0]['id']:
        return None

    complemento = cls(resultados[0])

    # ... procesamos los resultados ...
```

Procesando los Resultados del JOIN

```
for fila_en_db in resultados:
    # Verificamos si hay un taco asociado
    if fila_en_db['tacos.id']:
        # Parseamos los datos del taco
        datos_taco = {
            "id": fila_en_db['tacos.id'],
            "tortilla": fila_en_db['tortilla'],
            "guiso": fila_en_db['guiso'],
            "salsa": fila_en_db['salsa'],
            "restaurante_id": fila_en_db['restaurante_id'],
            "created_at": fila_en_db['tacos.created_at'],
            "updated_at": fila_en_db['tacos.updated_at'],
        }
        complemento.en_tacos.append(taco.Taco(datos_taco))

return complemento
```

Importante: Usamos `taco.Taco()` porque importamos el módulo, no la clase directamente.

Entendiendo el LEFT JOIN

La consulta SQL hace esto:

```
SELECT * FROM complementos
LEFT JOIN complementos_en_tacos ON complementos_en_tacos.complemento_id = complementos.id
LEFT JOIN tacos ON complementos_en_tacos.taco_id = tacos.id
WHERE complementos.id = 1;
```

Resultado: Una fila por cada taco relacionado con el complemento.

Si un complemento tiene 3 tacos, obtenemos **3 filas** con la misma información del complemento pero diferentes datos de tacos.

Columnas Duplicadas en el Resultado

Cuando hacemos JOIN, algunas columnas se repiten:

```
complementos.id | nombre_complemento | tacos.id | tortilla | guiso
----------------|--------------------|----------|----------|-------
      1         |     Cebolla        |    1     |  Maíz    | Asada
      1         |     Cebolla        |    2     |  Harina  | Pastor
```

Solución: Usar `tabla.columna` para distinguirlas:

- `complementos.id` → id del complemento
- `tacos.id` → id del taco

Extendiendo el Modelo Taco

Agregando Complementos al Modelo Taco

Modificamos `flask_app/models/taco.py`:

```
from flask_app.config.mysqlconnection import connectToMySQL
from flask_app.models import complemento  # Importamos el módulo

class Taco:
    def __init__(self, data):
        # ... atributos anteriores ...
        # Lista para almacenar los complementos relacionados
        self.complementos = []
```

Agregamos la lista `complementos` para almacenar los complementos asociados.

Método para Obtener Taco con Complementos

Agregamos el método al modelo Taco:

```
@classmethod
def get_taco_y_complementos(cls, datos):
    query = """
        SELECT * FROM tacos
        LEFT JOIN complementos_en_tacos ON complementos_en_tacos.taco_id = tacos.id
        LEFT JOIN complementos ON complementos_en_tacos.complemento_id = complementos.id
        WHERE tacos.id = %(id)s;
    """
    resultados = connectToMySQL('esquema_tacos').query_db(query, datos)

    if not resultados or not resultados[0]['id']:
        return None

    taco = cls(resultados[0])

    # ... procesamos los complementos ...
```

Procesando Complementos en el Taco

```
for fila_en_db in resultados:
    # Verificamos si hay un complemento asociado
    if fila_en_db['complementos.id']:
        datos_complemento = {
            "id": fila_en_db['complementos.id'],
            "nombre_complemento": fila_en_db['nombre_complemento'],
            "created_at": fila_en_db['complementos.created_at'],
            "updated_at": fila_en_db['complementos.updated_at'],
        }
        taco.complementos.append(complemento.Complemento(datos_complemento))

return taco
```

Ahora el objeto `taco` tiene su lista `complementos` llena con objetos `Complemento`.

⚠️ Importaciones Circulares

¿Qué es una Importación Circular?

Ocurre cuando dos módulos se importan mutuamente:

```
# complemento.py
from flask_app.models.taco import Taco  # ❌ Importa Taco

# taco.py
from flask_app.models.complemento import Complemento  # ❌ Importa Complemento
```

Problema: Python no puede resolver qué importar primero.

Solución: Importar el Módulo

En lugar de importar la clase, importamos el **módulo**:

```
# complemento.py
from flask_app.models import taco  # ✅ Importa el módulo

# taco.py
from flask_app.models import complemento  # ✅ Importa el módulo
```

Luego usamos `taco.Taco()` y `complemento.Complemento()`.

Comparación: Importación Directa vs Módulo

❌ Importación directa (circular):

```
from flask_app.models.taco import Taco
taco = Taco(datos)
```

✅ Importación de módulo (correcta):

```
from flask_app.models import taco
taco = taco.Taco(datos)
```

La segunda forma evita el problema de importación circular.

Mostrando en las Vistas

Modificando el Controlador

Actualizamos `flask_app/controllers/tacos.py`:

```
@app.route('/mostrar/<int:taco_id>')
def detalle(taco_id):
    datos = {
        'id': taco_id
    }
    # Usamos get_taco_y_complementos en lugar de get_one
    taco = Taco.get_taco_y_complementos(datos)
    return render_template("detalle.html", taco=taco)
```

Ahora el taco viene con sus complementos incluidos.

Actualizando la Plantilla HTML

```
<!-- Modificamos `flask_app/templates/detalle.html`: -->
<div class="card-body">
    <h2 class="card-title text-primary">Taco {{ taco.id }}</h2>
    <p><strong>Tortilla:</strong> {{ taco.tortilla }}</p>
    <p><strong>Guiso:</strong> {{ taco.guiso }}</p>
    <p><strong>Salsa:</strong> {{ taco.salsa }}</p>

    <hr>

    <h4>Complementos:</h4>
    {% if taco.complementos %}
        <ul class="list-group">
            {% for complemento in taco.complementos %}
                <li class="list-group-item">{{ complemento.nombre_complemento }}</li>
            {% endfor %}
        </ul>
    {% else %}
        <p class="text-muted">Este taco no tiene complementos asignados.</p>
    {% endif %}
</div>
```

Iterando sobre Complementos en Jinja2

Sintaxis:

```
{% for complemento in taco.complementos %}
    <li>{{ complemento.nombre_complemento }}</li>
{% endfor %}
```

Explicación:

- `taco.complementos` es la lista de objetos Complemento
- `complemento` es cada elemento en la iteración
- `complemento.nombre_complemento` accede al atributo del objeto

📋 Resumen de Conceptos Clave

Conceptos Clave de Muchos a Muchos

1. **Tabla intermedia:** Necesaria para relacionar dos entidades
2. **LEFT JOIN:** Consulta SQL para obtener datos relacionados
3. **Procesamiento de resultados:** Iterar sobre filas y construir objetos
4. **Importaciones circulares:** Evitar importando módulos, no clases
5. **Listas en modelos:** Almacenar objetos relacionados como atributos

Lo Más Importante

✅ Una relación muchos a muchos requiere una tabla intermedia

✅ LEFT JOIN nos permite obtener datos de múltiples tablas en una consulta

✅ Importar módulos (`import taco`) evita importaciones circulares

✅ Los resultados del JOIN vienen como múltiples filas, debemos procesarlas

✅ Las listas en `__init__` almacenan objetos relacionados

Recursos para Seguir Aprendiendo

- **Documentación MySQL:** https://dev.mysql.com/doc/
- **Flask SQLAlchemy:** Para ORM (más avanzado)
- **Relaciones en bases de datos:** Conceptos de modelado de datos
- **JOINs en SQL:** Práctica con diferentes tipos de JOIN

Próximos Pasos

En la siguiente clase aprenderemos:

- Autenticación y sesiones
- Validaciones de formularios
- Mensajes flash
- Protección de rutas

Práctica para Casa

Proyecto sugerido: Sistema de Biblioteca

- Libros y Autores (muchos a muchos)
- Un libro puede tener múltiples autores
- Un autor puede escribir múltiples libros
- Mostrar libros con sus autores y viceversa

Consejos Finales

- 💡 **Siempre verifica que existan datos** antes de procesar resultados del JOIN
- 💡 **Usa nombres descriptivos** para tablas intermedias (`complementos_en_tacos`)
- 💡 **Prueba tus consultas SQL** directamente en MySQL antes de implementarlas en Python
- 💡 **Mantén las importaciones consistentes** usando módulos para evitar problemas

🎉 ¡Felicidades!

Ya sabes trabajar con relaciones muchos a muchos

¡Has aprendido un concepto avanzado de bases de datos!

? Preguntas

¿Alguna duda sobre relaciones muchos a muchos?

🙌 ¡Excelente Trabajo!

¡Nos vemos en la siguiente clase con autenticación!

No olvides completar todos los ejercicios 📝