Laboratorio 1: Repaso de SQL

Condiciones del laboratorio

- Cada estudiante desarrolla el laboratorio de forma individual.
- Se recomienda el uso de su propio laptop si le es posible.
- El entregable del laboratorio se espera el 13 de Febrero 23:59.
- No requiere la instalación de bibliotecas adicionales a las incluidas en esta guía. Si su aplicación requiere bibliotecas adicionales, justifique ante el profesor antes de incluirlas.
- Puede utilizar herramientas de AI únicamente para la sección **D**. Si la usa, deberá escribirlo explícitamente en el documento de informe, y comunicarlo al profesor durante la sustentación.

A. Objetivos

- Crear la infraestructura en podman necesaria para una instancia de una BD.
- Repasar los conceptos básicos del lenguage SQL.
- Conectar desde Python a la base de datos para CRUD con SQLAlchemy ORM.

B. Preparación del laboratorio

A continuación, presentamos un listado de tareas requeridas para configurar la infrastructura necesaria para este laboratorio.

1. Podman Motor de Contenedores

Podman es un motor de contenedores, similar a *Docker* pero sin subproceso o *daemon*, para desarrollar, gestionar o ejecutar contenedores compatibles con OCI (Open Container Initiative). Podman tiene una interfaz de comandos CLI y una versión GUI llamada *Podman Desktop*

- Si aun no tiene instalado Podman, descarge e instale Podman de `https://podman.io/docs/installation`.
- Se recomienda utilizar Podman con WSL en vez de HyperThread para tener mejor rendimiento en Windows.
- (Opcional) Instale Podman Desktop 1.25.1 `https://podman-desktop.io/downloads/windows`
- Instancie una nueva MV de Podman, ejecute en terminal `podman machine init`.
- Inicie una VM, ejecute en terminal `podman machine start`.
- Lea el detalle de la documentación en https://docs.podman.io/en/latest/Commands.html.

2. Postgres en Podman

Las siguientes instrucciones crean un contenedor a partir de una imagen de PostgreSQL.

- Busque la imagen de PostgreSQL, ejecute `podman search postgres`
- Descargue la imagen de PostgreSQL, `podman pull docker.io/library/postgres`

El siguiente comando instancia un contenedor de PostgreSQL llamado *lab00-postgres*.

```
podman run -dt --name lab00-postgres -e POSTGRES_PASSWORD=1234 -p 5432:5432 -v "
<data_dir>:/var/lib/postgresql:Z" postgres
```

- Este contenedor tiene un nombre único *lab01-postgres*.
- Se declara una variable de ambiente con el password de la base de datos.
- El *volume* es un mapeo entre un directorio al interior del contenedor y un directorio en el host.
- Se puede conectar a la instancia de base de datos mediante una conexión al puerto 5432 mediante cualquier cliente SQL como Dbeaver https://dbeaver.io/.

3. Podman compose

Podman compose es un wrapper de `docker compose` con la Compose Specification que permite definir aplicaciones basadas en containers de forma agnóstica a la plataforma.
La especificación de *compose* la encuentra en https://github.com/compose-spec/compose-spec/blob/main/spec.md

Verifique que tenga instalado *compose*

```
podman compose version
```

Si obtiene un error, se puede activar el recurso *compose* desde Podman Desktop > Settings > Resources.

C. Trabajo de laboratorio

El laboratorio consiste en construir una aplicación simple de contenedores y soportada por una base de datos de PostgreSQL.

1. Estructura del proyecto

Construya una estructura de proyecto así:

```
lab01/
  compose.yaml
  db/init/
    01_schema.sql
    02_data.sql
```

Contenido del archivo `compose.yaml`:

```
services:
  db:
    image: postgres:18
    container_name: lab01_postgres
    environment:
      POSTGRES_PASSWORD: "1234"
      POSTGRES_USER: "postgres"
      POSTGRES_DB: "employees"
    ports:
      - "5432:5432"
    volumes:
      # Nombre del volume para persistencia
      - pg_data:/var/lib/postgresql
      # scripts iniciales
      - ./db/init:/docker-entrypoint-initdb.d:ro
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d employees"]
      interval: 5s
      timeout: 3s
      retries: 20

volumes:
  pg_data:
```

A continuación, explicamos el contenido del archivo de composición:

- Tenemos un servicio (por ahora) nombrado como *db* y construido a partir de la imagen `postgres:18`.
- Definimos variables de entorno para el password (para la forma segura use *secrets*), el nombre de la BD, y el usuario para conectarse posteriormente.
- Reenviamos el puerto 5432 para exponer el contenedor ante el host.
- Creamos un *volume* llamado *pg_data* que almacenará los datos de la BD.
- Mapeamos el *volume* al directorio `/var/lib/postgresql` del contenedor.
- Los scripts iniciales con el schema y los datos de la base de datos se cargarán desde el directorio `db/init/` al inicio de la instancia.
- Adicionalmente, tenemos un beat que comprueba la salud de la BD y demuestra si la base de datos está corriendo.

Contenido del esquema `01_schema.sql`:

```
CREATE TABLE IF NOT EXISTS department (
  dept_id SERIAL PRIMARY KEY,
  name TEXT NOT NULL UNIQUE
);

CREATE TABLE IF NOT EXISTS employees (
  emp_id SERIAL PRIMARY KEY,
  full_name TEXT NOT NULL,
  email TEXT UNIQUE,
  dept_id INT REFERENCES department(dept_id),
  hired_at DATE NOT NULL DEFAULT CURRENT_DATE
);
```

Contenido del archivo de datos `02_data.sql`:

```
INSERT INTO department (name) VALUES
('Engineering'),
('HR'),
('Finance')
ON CONFLICT (name) DO NOTHING;

INSERT INTO employees (full_name, email, dept_id, hired_at) VALUES
('Ana Ruiz', 'ana.ruiz@example.com', (SELECT dept_id FROM department WHERE
name='Engineering'), '2024-02-01'),
('Luis Perez', 'luis.perez@example.com', (SELECT dept_id FROM department WHERE
name='HR'), '2024-03-15'),
('Camila Gomez', 'camila.gomez@example.com', (SELECT dept_id FROM department WHERE
name='Finance'), '2024-05-10')
ON CONFLICT (email) DO NOTHING;
```

2. Ejecutar el compose

Luego, ejecute el siguiente comando en una terminal, y en el directorio de trabajo donde está el archivo *compose*.

```
podman compose up -d
```

up crea e inicia los contenedores.

-d: desacopla la terminal del proceso, y corre en background.

Luego, conectese a la base de datos mediante un cliente SQL. Consulte las tablas y el contenido.

```
select * from department d
```

3. Limpiar contenedores y volume

Cuando se crean los recursos de podman compose, se pueden eliminar todos los recursos (servicios) del compose. Ejecute en una terminal el siguiente comando:

```
podman compose down -v
```

-v: elimina los volumenes y los datos en él.

Si se inician los contenedores y volume de forma independiente, se pueden eliminar uno por uno, usando el nombre de los containers y los volume.

```
podman container rm lab01_postgres
podman volume rm pg_data
```

4. Agrege un contenedor con aplicación web

Ahora crearemos un nuevo contenedor con una aplicación web simple que muestra el contenido de las tablas de la base de datos. La aplicación, se conecta a la base de datos y permite filtrar por nombre de empleado.

La nueva estructura quedará así:

```
lab01/
  compose.yaml
  db/init/
    01_schema.sql
    02_data.sql
  web/
    app.py
    Dockerfile
    requirements.txt
```

El archivo `requirements.txt` contiene

```
Flask==3.0.3
SQLAlchemy==2.0.32
psycopg2-binary==2.9.9
gunicorn==22.0.0
```

El archivo `Dockerfile` de la aplicacion web contiene:

```
FROM python:3.11-slim
```

```
WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

EXPOSE 8000
CMD ["gunicorn", "-b", "0.0.0.0:8000", "app:app"]
```

Observe que:

- Se crea una instancia a partir de la imagen Python 3.11
- Se crea un directorio de trabajo `/app`.
- Se copia el archivo de requerimientos y se instalan.
- Se copia el archivo de la aplicación.
- Se expone el puerto 8000.
- Se lanza la aplicación HTTP mediante *gunicorn* como servidor WSGI (Web Server Gateway Interface).

El contenido del archivo `compose.yaml` se modificará de la siguiente forma:

```
services:
  db:
    image: postgres:18
    container_name: lab01_postgres
    environment:
      POSTGRES_PASSWORD: "1234"
      POSTGRES_USER: "postgres"
      POSTGRES_DB: "employees"
    ports:
      - "5432:5432"
    volumes:
      - pg_data:/var/lib/postgresql
      - ./db/init:/docker-entrypoint-initdb.d:ro
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d employees"]
      interval: 5s
      timeout: 3s
      retries: 20

  web:
    build: ./web
    container_name: lab01_web
    ports:
      - "8000:8000"
    depends_on:
      db:
        condition: service_healthy

volumes:
  pg_data:
```

El contenedor web, tiene su propio nombre, reenvío de puertos, y será instanciado posterior a la base de datos.

La aplicación web consiste en una instancia de Flask. Contenido del archivo `app.py`

```
import os
from flask import Flask, request, render_template_string
from sqlalchemy import create_engine, text

DATABASE_URL = os.environ.get("DATABASE_URL",
"postgresql+psycopg2://postgres:1234@db:5432/employees")

engine = create_engine(DATABASE_URL, pool_pre_ping=True, future=True)

app = Flask(__name__)

HTML = """
<!doctype html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>Empleados & Departamentos</title>
  <style>
    body { font-family: Arial, sans-serif; margin: 24px; }
    .box { border: 1px solid #ddd; padding: 16px; margin-bottom: 16px; border-
radius: 8px; }
    input, select { padding: 6px; }
    table { border-collapse: collapse; width: 100%; margin-top: 10px; }
    th, td { border: 1px solid #ddd; padding: 8px; text-align: left; }
    th { background: #f5f5f5; }
    .muted { color: #666; }
  </style>
</head>
<body>
  <h1>Empleados & Departamentos</h1>
  <p class="muted">Consultas realizadas en Postgres vía SQLAlchemy.</p>

  <div class="box">
    <h2>Departamentos</h2>
    <form method="get" action="/">
      <input type="hidden" name="view" value="departments"/>
      <button type="submit">Lista</button>
    </form>

    {% if view == "departments" %}
      <table>
        <tr><th>dept_id</th><th>name</th></tr>
        {% for r in departments %}
          <tr><td>{{ r.dept_id }}</td><td>{{ r.name }}</td></tr>
        {% endfor %}
      </table>
```

```
    {% endif %}
  </div>

  <div class="box">
    <h2>Empleados</h2>
    <form method="get" action="/">
      <input type="hidden" name="view" value="employees"/>
      <label>Departamento:</label>
      <select name="dept">
        <option value="">(todos)</option>
        {% for d in dept_options %}
          <option value="{{ d.name }}" {% if d.name == selected_dept %}selected{%
endif %}>{{ d.name }}</option>
        {% endfor %}
      </select>
      <label style="margin-left:10px;">Nombre contiene:</label>
      <input name="q" value="{{ q or '' }}" placeholder="Ejemplo: Ana"/>
      <button type="submit">Buscar</button>
    </form>

    {% if view == "employees" %}
      <table>
        <tr>
          <th>id_empleado</th><th>nombre</th><th>email</th><th>departamento</th>
<th>fecha_contratacion</th>
        </tr>
        {% for r in employees %}
          <tr>
            <td>{{ r.emp_id }}</td>
            <td>{{ r.full_name }}</td>
            <td>{{ r.email }}</td>
            <td>{{ r.department }}</td>
            <td>{{ r.hired_at }}</td>
          </tr>
        {% endfor %}
      </table>
      {% if employees|length == 0 %}
        <p class="muted">Sin resultados.</p>
      {% endif %}
    {% endif %}
  </div>

</body>
</html>
"""


@app.get("/")
def index():
    view = request.args.get("view", "")
    selected_dept = request.args.get("dept", "")
    q = request.args.get("q", "")

    with engine.connect() as conn:
```

```
        dept_options = conn.execute(text("SELECT name FROM department ORDER BY
name")).mappings().all()

        departments = []
        employees = []

        if view == "departments":
            departments = conn.execute(text("SELECT dept_id, name FROM department
ORDER BY dept_id")).mappings().all()

        if view == "employees":
            sql = """
              SELECT e.emp_id, e.full_name, e.email, d.name AS department,
e.hired_at
              FROM employees e
              LEFT JOIN department d ON d.dept_id = e.dept_id
              WHERE (:dept = '' OR d.name = :dept)
                AND (:q = '' OR LOWER(e.full_name) LIKE LOWER('%' || :q || '%'))
              ORDER BY e.emp_id;
            """
            employees = conn.execute(text(sql), {"dept": selected_dept or "", "q":
q or ""}).mappings().all()

    return render_template_string(
        HTML,
        view=view,
        departments=departments,
        employees=employees,
        dept_options=dept_options,
        selected_dept=selected_dept,
        q=q
    )

@app.get("/health")
def health():
    try:
        with engine.connect() as conn:
            conn.execute(text("SELECT 1"))
        return {"status": "ok"}
    except Exception as e:
        return {"status": "error", "detail": str(e)}, 500

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

Observe que:

- Se utiliza SQLAlchemy https://www.sqlalchemy.org/ como biblioteca para conectarse a la base de datos.
- Flask renderiza la plantilla Jinja mediante la función `render_template_string` que recibe los parámetros del string.

- Bajo la ruta de home `/` la aplicación muestra dos vistas, la consulta de departamentos y de los empleados.
- Las consultas se realizan directamente en lenguaje SQL. Posteriormente se reemplazarán las consultas de forma programática.
- Bajo la ruta `/health` el servicio responde un objeto JSON e informa si está activa la base de datos.
- Al interior del contenedor la aplicación Flask se expone en el puerto 8000.

Ejecute nuevamente el compose que crea e inicia los contenedores.

Consulte la página http://localhost:8000 desde su browser. Deberá ver contenido similar a la siguiente imagen.

Compruebe el comportamiento de la aplicación y observe los parámetros enviados a la plantilla para renderizar el contenido.

D. Sakila DVD Rental Store Chain

Sakila es una cadena de tiendas de alquiler de películas en DVD. Similar a Netflix cuando se dedicaba al alquiler de DVD por correo https://www.ebsco.com/research-starters/computer-science/netflix-inc .

Observe los datos de ejemplo `sakila_data.sql`:

1. Identifique las entidades, relaciones y atributos o campos desde el conjunto de datos.
2. Diseñe un esquema de base de datos con los elementos identificados.
3. En un archivo denominado `sakila_schema.sql` escriba el esquema que concuerde con `sakila_data.sql`.
4. Construya una estructura de proyecto como la mostrada en el punto anterior con `compose.yaml` y los respectivos contenedores.
5. Cargue los datos del ejemplo Sakila del archivo `sakila_data.sql` de las respectivas tablas.
6. Escriba las consultas que contesten a las siguientes preguntas de negocio:
    1. ¿Cuáles son las películas con más alquileres por categoría?
    2. ¿Cuáles son los clientes que su gasto total es superior al promedio?
    3. ¿Cuáles son las películas más alquiladas que el promedio de su categoría?
    4. ¿Cuáles son los clientes que alquilaron en el primer trimestre pero no en el segundo?
7. Investigue sobre el uso de SQLAlchemy como ORM en vez de usar SQL directamente en el código.
8. Implemente la solución a las preguntas anteriores en SQLAlchemy ORM.

E. Entregables

1. Carga al repositorio

El entregable del laboratorio se debe cargar al repositorio del curso que contiene:

1. Código implementado. Archivos `01_schema.sql`,

```
lab01/
  compose.yaml
  db/init/
    01_schema.sql
    02_data.sql
  web/
    app.py
    Dockerfile
    model.py
    queries.py
    requirements.txt
```

2. Archivo con el diseño del esquema de base de datos https://www.drawio.com/, en https://dbdiagram.io/, o directamente en el documento si usa *mermaid* o *startuml*.
3. Archivo Markdown, documento de solución a las preguntas de negocio.

2. Sustentación del laboratorio

Durante la sustentación, el estudiante presenta su trabajo y modifica el entregable para responder a un nuevo requerimiento o pregunta de negocio.

Ejemplo de pregunta de negocio:

- ¿Cuál es el impacto en la facturación de las devoluciones tardias por categoría y segmento del cliente?