# SQL With Flask

## Basics of SQL

**SQL Basics in Command Prompt**

## 1. Introduction to SQL in Command Prompt

SQL (Structured Query Language) is used for managing and querying relational databases. SQL Server provides `sqlcmd`, a command-line tool for executing SQL commands.

## 2. Types of SQL Commands

SQL commands are categorized into different types based on their functionality:

### Data Definition Language (DDL)

DDL commands are used to define and modify database structures.

- `CREATE` – Creates a new database, table, or other objects.
- `ALTER` – Modifies an existing database object.
- `DROP` – Deletes a database or table.
- `TRUNCATE` – Removes all records from a table without logging individual row deletions.

### Data Manipulation Language (DML)

DML commands are used to modify data stored in the database.

- `INSERT` – Adds new records to a table.
- `UPDATE` – Modifies existing records.
- `DELETE` – Removes specific records from a table.

### Data Query Language (DQL)

DQL is used to retrieve data from the database.

- `SELECT` – Retrieves data from one or more tables.

### Data Control Language (DCL)

DCL commands manage permissions and access control.

- `GRANT` – Provides specific privileges to users.
- `REVOKE` – Removes specific privileges from users.

## 3. SQL Data Types

SQL supports various data types categorized as follows:

### Numeric Data Types:

- `INT` – Integer values.
- `BIGINT` – Large integer values.
- `SMALLINT` – Small integer values.
- `DECIMAL(p,s)` – Fixed precision and scale decimal numbers.
- `FLOAT` – Approximate floating-point numbers.
- `REAL` – Single-precision floating-point numbers.

### String Data Types:

- `CHAR(n)` – Fixed-length character data.
- `VARCHAR(n)` – Variable-length character data.
- `TEXT` – Large character data.

### Date and Time Data Types:

- `DATE` – Stores date values.
- `DATETIME` – Stores date and time values.
- `TIME` – Stores only time values.
- `TIMESTAMP` – Stores automatic timestamps.

### Boolean Data Type:

- `BIT` – Stores 1 (TRUE) or 0 (FALSE).

### Other Data Types:

- `BLOB` – Stores binary large objects such as images or files.
- `JSON` – Stores JSON-formatted data.
- `XML` – Stores XML-formatted data.

## 4. Connecting to SQL Server via Command Prompt

1. Open Command Prompt (`cmd`).
2. Connect to the SQL Server instance using:

   ```
   sqlcmd -S server_name -U username -P password
   ```

   Replace `server_name`, `username`, and `password` with your actual SQL Server details.
3. To connect using Windows authentication:

   ```
   sqlcmd -S server_name -E
   ```

## 5. Basic SQL Commands in Command Prompt

### Checking Available Databases

```
SELECT name FROM sys.databases;
GO
```

### Creating a Database

```
CREATE DATABASE TestDB;
GO
```

## Using a Database

```
USE TestDB;
GO
```

## Creating a Table

```
CREATE TABLE Users (
    id INT PRIMARY KEY IDENTITY(1,1),
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL
);
GO
```

## Inserting Data

```
INSERT INTO Users (name, email, password_hash)
VALUES ('John Doe', 'john@example.com', 'hashed_password');
GO
```

## Retrieving Data

```
SELECT * FROM Users;
GO
```

## Using the WHERE Clause

```
SELECT * FROM Users WHERE email = 'john@example.com';
GO
```

## Sorting Data

```
SELECT * FROM Users ORDER BY name ASC;
GO
```

## Limiting Results

```
SELECT TOP 5 * FROM Users;
GO
```

## Aggregating Data

```
SELECT COUNT(*) FROM Users;
GO
```

## Using GROUP BY

```
SELECT name, COUNT(*) AS user_count FROM Users GROUP BY name;
GO
```

## Using HAVING

```
SELECT name, COUNT(*) AS user_count FROM Users GROUP BY name
HAVING COUNT(*) > 1;
GO
```

## Using ORDER BY

```
SELECT * FROM Users ORDER BY email DESC;
GO
```

## Updating Data

```
UPDATE Users
SET email = 'john.doe@example.com'
WHERE id = 1;
GO
```

### Deleting Data

```
DELETE FROM Users WHERE id = 1;
GO
```

### Dropping a Table

```
DROP TABLE Users;
GO
```

### Dropping a Database

```
DROP DATABASE TestDB;
GO
```

## 6. Exiting SQLCMD

To exit `sqlcmd`, type:

```
EXIT
```

## SQL with FLASK

## 1. Introduction to SQL Server

SQL Server is a relational database management system (RDBMS) developed by Microsoft. It is used for storing and managing data efficiently.

### Installing SQL Server

1. Download and install SQL Server from the official Microsoft website.
2. Install SQL Server Management Studio (SSMS) for managing databases.
3. Open SSMS and connect to the SQL Server instance.

### Creating a Database in SQL Server

1. Open SSMS and connect to the server.
2. In the Object Explorer, right-click on "Databases" > "New Database".
3. Enter a database name (e.g., `FlaskAppDB`) and click "OK".
4. Use the following SQL script to create a table:

   ```
   CREATE TABLE Users (
       id INT PRIMARY KEY IDENTITY(1,1),
       name VARCHAR(100) NOT NULL,
       email VARCHAR(100) UNIQUE NOT NULL,
       password_hash VARCHAR(255) NOT NULL
   );
   ```

# Connecting to a Database Using PyODBC

## Introduction

PyODBC is a Python library that provides an interface for connecting to databases using the Open Database Connectivity (ODBC) standard. This guide will show how to install PyODBC, connect to an SQL Server database, and perform basic CRUD operations.

---

## Step 1: Install PyODBC

Before using PyODBC, install it via pip:

```
pip install pyodbc
```

---

## Step 2: Establish a Connection

Create a Python script (e.g., `database.py`) and import PyODBC:

```
import pyodbc

# Define connection string
conn = pyodbc.connect(
    "DRIVER={ODBC Driver 17 for SQL Server};"
    "SERVER=your_server;"
    "DATABASE=your_database;"
    "UID=your_username;"
    "PWD=your_password;"
    "Trusted_Connection=yes;"
)

# Create a cursor
cursor = conn.cursor()
```

### Explanation:

- `DRIVER`: Specifies the ODBC driver for SQL Server (ensure it's installed on your system).
- `SERVER`: The database server address.
- `DATABASE`: The name of the database.
- `UID` and `PWD`: The username and password for authentication.

---

## Step 3: Create a Table

Use the following SQL command to create a table:

```
cursor.execute('''
    CREATE TABLE Users (
        ID INT PRIMARY KEY IDENTITY(1,1),
        Name VARCHAR(100),
        Email VARCHAR(100) UNIQUE
    )
''')
conn.commit()
```

### Explanation:

- `ID`: Auto-incrementing primary key.
- `Name`: Stores user names.
- `Email`: Stores unique email addresses.

---

## Step 4: Insert Data

To insert a new record:

```
cursor.execute("INSERT INTO Users (Name, Email) VALUES (?,
?)", ("John Doe", "john@example.com"))
conn.commit()
```

### Explanation:

- `?`: Placeholder for parameterized queries to prevent SQL injection.
- `conn.commit()`: Saves the changes.

---

## Step 5: Read Data

To fetch and display all users:

```
cursor.execute("SELECT * FROM Users")
for row in cursor.fetchall():
    print(row)
```

### Explanation:

- `fetchall()`: Retrieves all records from the table.

---

## Step 6: Update Data

Modify an existing record:

```
cursor.execute("UPDATE Users SET Name = ? WHERE ID = ?", ("Ja
ne Doe", 1))
conn.commit()
```

### Explanation:

- Updates the `Name` field where `ID` matches 1.

---

## Step 7: Delete Data

Remove a record:

```
cursor.execute("DELETE FROM Users WHERE ID = ?", (1))
conn.commit()
```

### Explanation:

- Deletes the record where `ID` is 1.

---

## Step 8: Close the Connection

After all operations, close the connection:

```
cursor.close()
conn.close()
```

---

## Conclusion

This guide demonstrated how to connect to a database using PyODBC, execute SQL commands, and perform CRUD operations. You can extend this by integrating with Flask or using advanced queries.

# Using SQLAlchemy to Connect a Flask App to a Database

## Introduction

SQLAlchemy is a popular ORM (Object-Relational Mapping) tool for Python that simplifies database interactions. In this guide, we will create a simple Flask app, connect it to an SQLite database using SQLAlchemy, and implement CRUD (Create, Read, Update, Delete) operations.

---

## Step 1: Install Dependencies

Ensure you have Flask and SQLAlchemy installed:

```
pip install flask_sqlalchemy
```

---

## Step 2: Setting Up Flask and SQLAlchemy

Create a file named `app.py` and set up Flask with SQLAlchemy:

```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'mssql+pyodbc://DESKT
OP-TJABPSR/db?driver=ODBC+Driver+17+for+SQL+Server'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

```
db = SQLAlchemy(app)
app.app_context().push()
```

### Explanation:

- `SQLALCHEMY_DATABASE_URI`: Specifies the database connection. You can replace it with a PostgreSQL or MySQL URI.
- `SQLALCHEMY_TRACK_MODIFICATIONS`: Disables tracking modifications to improve performance.

---

## Step 3: Creating a Model

Define a database model to represent an entity:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=F
alse)

    def to_dict(self):
        return {'id': self.id, 'name': self.name, 'email': se
lf.email}
```

### Explanation:

- `id`: Primary key, auto-incremented.
- `name`: A required string field.
- `email`: A unique, required string field.
- `to_dict()`: Helper function to convert a model instance into a dictionary.

Run the following command in a Python shell to create the database:

```
from app import db

db.create_all()
```

---

## Step 4: Implementing CRUD Operations

### 1. Create a New User (POST)

```
@app.route('/users', methods=['POST'])
def create_user():
    data = request.json
    new_user = User(name=data['name'], email=data['email'])
    db.session.add(new_user)
    db.session.commit()
    return jsonify(new_user.to_dict()), 201
```

### Explanation:

- Extracts JSON data from the request.
- Creates a new `User` instance.
- Adds it to the session and commits it to the database.

### 2. Retrieve All Users (GET)

```
@app.route('/users', methods=['GET'])
def get_users():
    users = User.query.all()
    return jsonify([user.to_dict() for user in users])
```

### Explanation:

- Queries all user records.
- Converts them into a list of dictionaries and returns them as JSON.

## 3. Retrieve a Single User by ID (GET)

```
@app.route('/users/<int:id>', methods=['GET'])
def get_user(id):
    user = User.query.get_or_404(id)
    return jsonify(user.to_dict())
```

## Explanation:

- Retrieves a user by ID.
- Returns a 404 error if the user does not exist.

## 4. Update a User (PUT)

```
@app.route('/users/<int:id>', methods=['PUT'])
def update_user(id):
    user = User.query.get_or_404(id)
    data = request.json
    user.name = data.get('name', user.name)
    user.email = data.get('email', user.email)
    db.session.commit()
    return jsonify(user.to_dict())
```

## Explanation:

- Retrieves the user by ID.
- Updates the fields with new data if provided.
- Commits changes to the database.

## 5. Delete a User (DELETE)

```
@app.route('/users/<int:id>', methods=['DELETE'])
def delete_user(id):
    user = User.query.get_or_404(id)
    db.session.delete(user)
    db.session.commit()
    return jsonify({'message': 'User deleted'}), 200
```

### Explanation:

- Retrieves the user by ID.
- Deletes the user from the database.
- Returns a success message.

---

## Step 5: Running the Flask App

Run the app using:

```
flask run
```

By default, Flask runs on `http://127.0.0.1:5000/`.

---

## Step 6: Testing the API Using Postman

Use **Postman** to test the API by following these steps:

### 1. Create a User (POST)

- Open Postman
- Select **POST**
- Enter URL: `http://127.0.0.1:5000/users`
- Go to the **Body** tab, select **raw**, and choose **JSON** format
- Enter the following JSON:

  ```
  {
    "name": "John Doe",
    "email": "john@example.com"
  }
  ```

- Click **Send**

## 2. Retrieve All Users (GET)

- Select **GET**
- Enter URL: `http://127.0.0.1:5000/users`
- Click **Send**

## 3. Retrieve a Single User by ID (GET)

- Select **GET**
- Enter URL: `http://127.0.0.1:5000/users/1`
- Click **Send**

## 4. Update a User (PUT)

- Select **PUT**
- Enter URL: `http://127.0.0.1:5000/users/1`
- Go to the **Body** tab, select **raw**, and choose **JSON** format
- Enter the following JSON:

  ```
  {
    "name": "Jane Doe"
  }
  ```

- Click **Send**

## 5. Delete a User (DELETE)

- Select **DELETE**
- Enter URL: `http://127.0.0.1:5000/users/1`
- Click **Send**

---