1-se

give me a bootstrap html form with 3 fields and 2 buttons.

Here’s a simple Bootstrap 5 form with three fields (Name, Email, Message) and two buttons (Submit and Reset):

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Bootstrap Form Example</title>
  <!-- Bootstrap CSS -->
  <link
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
    rel="stylesheet"
  />
</head>
<body>
  <div class="container mt-5">
    <form>
      <!-- Field 1: Name -->
      <div class="mb-3">
        <label for="inputName" class="form-label">Name</label>
        <input
          type="text"
          class="form-control"
          id="inputName"
          placeholder="Enter your name"
        />
      </div>

      <!-- Field 2: Email -->
      <div class="mb-3">
        <label for="inputEmail" class="form-label">Email address</label>
        <input
          type="email"
          class="form-control"
          id="inputEmail"
          placeholder="name@example.com"
        />
      </div>

      <!-- Field 3: Message -->
      <div class="mb-3">
        <label for="inputMessage" class="form-label">Message</label>
        <textarea
          class="form-control"
          id="inputMessage"
          rows="3"
          placeholder="Type your message here"
        ></textarea>
      </div>

      <!-- Buttons -->
      <button type="submit" class="btn btn-primary me-2">Submit</button>
      <button type="reset" class="btn btn-secondary">Reset</button>
    </form>
  </div>
  <!-- Bootstrap JS (optional) -->
  <script
    src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"
  ></script>
</body>
</html>
```

Notes:

- Uses Bootstrap's CDN for quick setup.
- `mb-3` adds spacing between form groups.
- The two buttons are styled with `btn-primary` (Submit) and `btn-secondary` (Reset).

python SqlDataService() delete by id methods

Below is an example of a Python "`SqlDataService`" class using SQLAlchemy (sync) that provides generic delete-by-id methods. You can adapt it to your own session/engine setup, or swap in raw SQL if you prefer.

python

```python
from typing import Type, List, Union
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, Session
from sqlalchemy.exc import NoResultFound
from sqlalchemy.ext.declarative import declarative_base

# Base class for declarative models
Base = declarative_base()

class SqlDataService:
    def __init__(self, connection_string: str):
        """
        Initialize the data service.

        :param connection_string: e.g., "postgresql://user:pass@host/dbname" or
"sqlite:///my.db"
        """
        self.engine = create_engine(connection_string, echo=False, future=True)
        self.SessionLocal = sessionmaker(bind=self.engine, autoflush=False, autocommit=False,
future=True)

    def delete_by_id(self, model: Type[Base], id_value: Union[int, str]) -> bool:
        """
        Delete a single row from `model` table by its primary-key value.

        :param model: Declarative model class
        :param id_value: Primary key of the row to delete
        :return: True if a row was deleted, False if no such row existed
        """
        with self.SessionLocal() as session:
            stmt = session.query(model).filter(model.id == id_value)
            count = stmt.delete(synchronize_session="fetch")
            session.commit()
            return bool(count)

    def delete_by_ids(self, model: Type[Base], id_list: List[Union[int, str]]) -> int:
        """
        Delete multiple rows from `model` whose primary keys are in `id_list`.

        :param model: Declarative model class
        :param id_list: List of primary-key values
        :return: Number of rows deleted
        """
        if not id_list:
            return 0

        with self.SessionLocal() as session:
            stmt = session.query(model).filter(model.id.in_(id_list))
            count = stmt.delete(synchronize_session="fetch")
            session.commit()
            return count

    def delete_or_raise(self, model: Type[Base], id_value: Union[int, str]) -> None:
        """
        Delete a row by id, but raise if it doesn’t exist.

        :param model: Declarative model class
        :param id_value: Primary key
        :raises NoResultFound: if no row with given id
        """
        with self.SessionLocal() as session:
            obj = session.get(model, id_value)
            if obj is None:
                raise NoResultFound(f"{model.__name__} with id={id_value} not found")
            session.delete(obj)
            session.commit()
```

How to use

1. **Define your model**:

    ```
    python

    from sqlalchemy import Column, Integer, String

    class User(Base):
        __tablename__ = 'users'
        id = Column(Integer, primary_key=True, index=True)
        name = Column(String, nullable=False)
    ```

2. **Create tables (once)**:

    ```
    python

    service = SqlDataService("sqlite:///./test.db")
    Base.metadata.create_all(service.engine)
    ```

3. **Delete operations**:

    ```
    python

    service = SqlDataService("sqlite:///./test.db")

    # Delete user with id=5; returns True if deleted, False if not found
    deleted = service.delete_by_id(User, 5)
    print("Deleted?", deleted)

    # Delete several users at once
    count = service.delete_by_ids(User, [2, 3, 7])
    print(f"Deleted {count} users")

    # Delete or raise if missing
    try:
        service.delete_or_raise(User, 42)
    ```

```
except NoResultFound as e:
    print("Oops:", e)
```

Notes & alternatives

- You can swap in raw SQL using `session.execute("DELETE FROM ... WHERE id=:id", {"id": id_value})` if you don't use the ORM.
- `synchronize_session="fetch"` keeps the session state in sync; you can also use `"evaluate"` or leave it out for performance if you know the session won't reuse those objects.
- For async / `asyncpg` / `SQLAlchemy 2.0 async API`, the pattern is very similar but you'd use an `AsyncSession` and `await` your calls.

give me delete sql query

Here are the most common forms of a SQL DELETE query:

```sql
-- 1. Delete a single row by its primary-key (id):
DELETE FROM table_name
WHERE id = 123;


-- 2. Delete multiple rows whose ids are in a given list:
DELETE FROM table_name
WHERE id IN (101, 102, 103);


-- 3. Delete rows matching some other condition (e.g. older than a date):
DELETE FROM table_name
WHERE created_at < '2025-01-01';


-- 4. Delete all rows in the table (use with extreme caution!):
DELETE FROM table_name;
```

Notes:

- Always include a **WHERE** clause to avoid wiping out your entire table unintentionally.
- Wrap your DELETEs in a transaction if your database supports it, so you can roll back on mistakes:

  ```sql
  BEGIN;
    DELETE FROM table_name WHERE id = 123;
  -- ROLLBACK;  -- if something went wrong
  COMMIT;
  ```

- In many dialects (PostgreSQL, MySQL, SQL Server, Oracle), parameterize your queries when executing from code, e.g.:

```
sql

DELETE FROM table_name WHERE id = ?;
```

or

```
sql

DELETE FROM table_name WHERE id = :id;
```

to safely bind your variable.

give me sql update query

Here are the most common patterns for an SQL UPDATE statement:

```
sql

-- 1. Update a single column on one row by its primary key:
UPDATE table_name
SET    column1 = 'new_value'
WHERE  id = 123;


-- 2. Update multiple columns on one row:
UPDATE table_name
SET    column1 = 'new_value',
       column2 = 42,
       column3 = NULL
WHERE  id = 123;


-- 3. Update many rows matching a condition:
UPDATE table_name
SET    status = 'archived'
WHERE  created_at < '2024-01-01';


-- 4. Update using values from another table (ANSI-SQL JOIN style):
UPDATE t
SET    t.status       = s.new_status,
       t.last_updated = CURRENT_TIMESTAMP
FROM   table_name       AS t
JOIN   status_updates   AS s
  ON   t.id = s.target_id
WHERE  s.effective_date <= CURRENT_DATE;


-- 5. Update with a subquery:
UPDATE table_name
SET    score = (
         SELECT AVG(rating)
         FROM   reviews
         WHERE  reviews.item_id = table_name.id
       )
WHERE  EXISTS (
         SELECT 1
         FROM   reviews
         WHERE  reviews.item_id = table_name.id
       );


-- 6. Update all rows (no WHERE — use with caution!):
```

```
UPDATE table_name
SET    is_active = FALSE;
```

Best Practices

- **Always include a WHERE clause** (unless you really mean “all rows”) to avoid unintentionally wiping out your data.
- **Wrap in a transaction** so you can roll back if something goes wrong:

  ```
  sql

  BEGIN;
    UPDATE table_name
    SET    column1 = 'value'
    WHERE  id = 123;
  -- ROLLBACK;  -- if you detect an issue
  COMMIT;
  ```

- **Parameterize** your queries when running them from application code to protect against SQL injection:

  ```
  sql

  -- e.g. in Python psycopg2 or similar
  cursor.execute(
    "UPDATE table_name SET column1 = %s WHERE id = %s",
    (new_value, record_id)
  )
  ```

- **Return affected data** (if your database supports it) to see what was changed:

  ```
  sql

  -- PostgreSQL
  UPDATE table_name
  SET    column1 = 'new'
  WHERE  id = 123
  RETURNING *;
  ```

- **Check the row count** your driver returns to verify how many records were updated.