

To this form now I must to backend connection and store data entered in db What is the process using python and FastAPI And node postgres

ChatGPT said:

To store the submitted form data into a database, you need a **backend API + database connection**. I'll explain **both options you mentioned**:


✓ Option 1 — FastAPI (Python) + PostgreSQL

Requirements

Install packages:

```
pip install fastapi uvicorn sqlalchemy psycopg2 pydantic
```

Step 1 — Database model (SQLAlchemy)

 database.py

```
from sqlalchemy import create_engine  
from sqlalchemy.orm import sessionmaker, declarative_base
```


```
DATABASE_URL = "postgresql://postgres:<password>@localhost:5432/<dbname>"
```

```
engine = create_engine(DATABASE_URL)
```

```
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
```

```
Base = declarative_base()
```

Step 2 — Create table model

 models.py

```
from sqlalchemy import Column, Integer, String, Float  
from database import Base
```

```
class LostForm(Base):
```

```
    __tablename__ = "lost_forms"
```

```
    id = Column(Integer, primary_key=True, index=True)
```


```
sno = Column(Integer)
tenderName = Column(String)
customer = Column(String)
tenderType = Column(String)
documentType = Column(String)
valueWithoutGST = Column(Float)
valueWithGST = Column(Float)
reasonForLosing = Column(String)
yearLost = Column(Integer)
partners = Column(String)
competitors = Column(String)
technicalScore = Column(Float)
quotedPrice = Column(Float)
```

Run **table migration** to create DB:

```
from database import engine
from models import Base
```

```
Base.metadata.create_all(bind=engine)
```

Step 3 — FastAPI endpoint

 main.py

```
from fastapi import FastAPI, Depends
from sqlalchemy.orm import Session
from database import SessionLocal
from models import LostForm
```

```
app = FastAPI()
```

```
def get_db():
    db = SessionLocal()
    try:
```

```

        yield db

    finally:
        db.close()

@app.post("/lost")
def submit_lost_form(data: dict, db: Session = Depends(get_db)):
    lost = LostForm(**data) # auto maps JSON → DB model
    db.add(lost)
    db.commit()
    db.refresh(lost)
    return {"status": "success", "saved_id": lost.id}

```

🔴 Step 4 — Frontend request from React

Instead of snackbar-only submit, add API call:


```

const onSubmit = async (data) => {
  try {
    const response = await fetch("http://localhost:8000/lost", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(data),
    });
    if (response.ok) {
      setSubmitSuccess(true);
      reset();
    }
  } catch (error) {
    console.error("Error saving data:", error);
  }
};

```

► **Start FastAPI server**

```
uvicorn main:app --reload
```

 When you click **Submit**, the form will now store data in **PostgreSQL**.

✓ Option 2 — Node.js + Express + PostgreSQL

Requirements

```
npm install express pg cors body-parser
```

Step 1 — DB connection


 db.js

```
const { Pool } = require("pg");
```

```
const pool = new Pool({  
  user: "postgres",  
  host: "localhost",  
  database: "<dbname>",  
  password: "<password>",  
  port: 5432,  
});
```

```
module.exports = pool;
```

Step 2 — API endpoint

 server.js

```
const express = require("express");
```

```
const pool = require("./db");
```

```
const cors = require("cors");
```

```
const app = express();
```

```
app.use(cors());
```

```
app.use(express.json());
```

```

app.post("/lost", async (req, res) => {
  const data = req.body;

  const query = `
    INSERT INTO lost_forms (
      slno, tendertype, customer, tendertype, documenttype,
      valuetype, valuetype, reasonforlosing,
      yearlost, partners, competitors, technicalscore, quotedprice
    ) VALUES ($1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13) RETURNING id;
  `;

  const values = [
    data.slno, data.tenderName, data.customer, data.tenderType, data.documentType,
    data.valueWithoutGST, data.valueWithGST, data.reasonForLosing,
    data.yearLost, data.partners, data.competitors, data.technicalScore, data.quotedPrice
  ];

  const result = await pool.query(query, values);
  res.json({ status: "success", saved_id: result.rows[0].id });
});

app.listen(5000, () => console.log("Server running on 5000"));

```

✦ Step 3 — React connects same as FastAPI

```

await fetch("http://localhost:5000/lost", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(data),
});

```

★ Recommendation

Stack	When to use
FastAPI	Best for Python-based ML/analytics projects
Node.js	Best when frontend team and backend both use JavaScript
Both + PostgreSQL Reliable, scalable & enterprise ready	

If you want next:

I can provide complete pack:

- ◆ Ready PostgreSQL .sql schema
- ◆ Auto-table migration script
- ◆ Docker setup for DB + backend
- ◆ Authentication (login + token)
- ◆ Uploading Corrigendum / LOI file to backend storage

Just say “**continue**” and mention **FastAPI or Node**. 🚀

You said:

Using python and fastAPI give me steps to do for connection

ChatGPT said:

Below is a **clear step-by-step guide** to connect your React form to **Python FastAPI with PostgreSQL** and **store form data**.

✅ Step 1 — Create a folder structure

backend/

```
|— main.py
|— database.py
|— models.py
|— schemas.py
|— requirements.txt
```

✅ Step 2 — Install required dependencies

Run this in your backend folder:

```
pip install fastapi uvicorn sqlalchemy psycpg2-binary pydantic
```

Add to requirements.txt (optional for pip install later):

fastapi

uvicorn

sqlalchemy

psycopg2-binary

pydantic

✅ Step 3 — Configure PostgreSQL database connection

📁 database.py

```
from sqlalchemy import create_engine
```

```
from sqlalchemy.orm import sessionmaker, declarative_base
```

```
DATABASE_URL = "postgresql://postgres:<PASSWORD>@localhost:5432/<DBNAME>"
```

```
engine = create_engine(DATABASE_URL)
```

```
SessionLocal = sessionmaker(bind=engine, autocommit=False, autoflush=False)
```

```
Base = declarative_base()
```

- ◆ Replace <PASSWORD> and <DBNAME> with your actual values.
-

✅ Step 4 — Create the table model (ORM)

📁 models.py

```
from sqlalchemy import Column, Integer, String, Float
```

```
from database import Base
```

```
class LostForm(Base):
```

```
    __tablename__ = "lost_forms"
```

```
    id = Column(Integer, primary_key=True, index=True)
```

```
    slno = Column(Integer)
```

```
    tenderName = Column(String)
```

```
customer = Column(String)
tenderType = Column(String)
documentType = Column(String)
valueWithoutGST = Column(Float)
valueWithGST = Column(Float)
reasonForLosing = Column(String)
yearLost = Column(Integer)
partners = Column(String)
competitors = Column(String)
technicalScore = Column(Float)
quotedPrice = Column(Float)
```

✅ Step 5 — Create the schema (data validation)

📁 schemas.py

```
from pydantic import BaseModel
```

```
class LostFormSchema(BaseModel):
    slno: int
    tenderName: str
    customer: str
    tenderType: str
    documentType: str
    valueWithoutGST: float
    valueWithGST: float
    reasonForLosing: str
    yearLost: int
    partners: str | None = None
    competitors: str | None = None
    technicalScore: float | None = None
    quotedPrice: float | None = None
```



```
class Config:

    orm_mode = True
```

✅ Step 6 — Create FastAPI endpoint to save form data

📁 main.py

```
from fastapi import FastAPI, Depends
from sqlalchemy.orm import Session
from database import SessionLocal, engine
from models import Base, LostForm
from schemas import LostFormSchema

# Create tables in DB
Base.metadata.create_all(bind=engine)

app = FastAPI()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.post("/lost")
def create_lost_form(form: LostFormSchema, db: Session = Depends(get_db)):
    lost_data = LostForm(**form.dict())
    db.add(lost_data)
    db.commit()
    db.refresh(lost_data)
    return {"message": "Saved successfully", "id": lost_data.id}
```

✅ Step 7 — Run FastAPI server

uvicorn main:app --reload

API will run at:

http://localhost:8000

Docs with testing UI at:

http://localhost:8000/docs

🔗 Step 8 — Connect React form to FastAPI

Inside your React form onSubmit function:

```
const onSubmit = async (data) => {  
  try {  
    const res = await fetch("http://localhost:8000/lost", {  
      method: "POST",  
      headers: { "Content-Type": "application/json" },  
      body: JSON.stringify(data),  
    });  
  
    if (res.ok) {  
      setSubmitSuccess(true); // existing snackbar  
      reset();  
    }  
  } catch (error) {  
    console.error("Error:", error);  
  }  
};
```

✓ Now when user clicks **Submit**, the data is stored in **PostgreSQL**.





🚀 Optional Add-ons I can provide if you need

Feature

Status

File uploading (Corrigendum / LOI)

➡️ SOON Available

Feature	Status
JWT login authentication	 <small>SOON</small> Available
Save audit history / timestamps	 <small>SOON</small> Available
Export DB to Excel / PDF	 <small>SOON</small> Available
Deployment guide (Render / AWS / Azure)	 <small>SOON</small> Available

If you want next:

Tell me YES and I can generate:

- ◆ PostgreSQL .sql file for auto table creation
- ◆ Full **backend folder zip** with ready code
- ◆ **MongoDB version** of the same API
- ◆ **Docker setup** for backend + PostgreSQL

Just say “**continue**” 

You said:

How is the microservice approach I have one form that works independently The database is different for each form?


ChatGPT said:

Great question — this is exactly where microservice architecture becomes useful.

Microservice Architecture Concept for Your Case

You currently have:

- **Multiple forms (Form 1, Form 2, ... Form 7 etc.)**
- Each form **works independently**
- Requirement: **Each form stores data separately**

 This aligns **very well** with a **microservice approach**.

What is a microservice in your scenario?

A **microservice** = **one independent backend module** responsible for:

Form	Microservice	DB Table
Form 1	Service 1	Table 1
Form 2	Service 2	Table 2
Form 3	Service 3	Table 3
Form 7 (LOST Form)	Lost Service	lost_forms table
Form 6 (Order Received)	Orders Service	order_received table
...

Each service is:

- ◆ Independent code
- ◆ Independent database model
- ◆ Independent API endpoint
- ◆ Deployable separately

🔥 Why microservices makes sense for your forms?

Without Microservices (Single Backend) With Microservices

One failure affects all forms	Failure affects only that form
Big codebase	Small independent codebases
Hard to scale	Scale only the busy form
Slow deployments	Can deploy form-by-form
Same database for all	Each form can have its own DB

→ Works better as your number of forms increases.

📌 Database decision in microservices

Microservices give **two choices**:

✓ Option A — Each microservice has its own separate database

- ◆ Database scaling per form
- ◆ No cross-impact on other forms
- ◆ Best isolation & clean boundaries

Example:

Form	Database
------	----------

Form 7 (LOST)	lost_db
---------------	---------

Form 6 (Order Received)	orders_db
-------------------------	-----------

Form 5 (CRM Leads)	crm_db
--------------------	--------

Best for enterprise projects — but requires more database admin.

✓ Option B — All microservices share one database, but separate tables

Example DB:

| lost_forms | order_received | crm_leads | ... |

Easier to maintain in small teams, also works fine.

🌟 Recommendation based on your system size

Project Size	Suggested DB Architecture
--------------	---------------------------

< 10 services, small team	One DB → multiple tables
---------------------------	--------------------------

> 10 services, serious enterprise	Separate DB per microservice
-----------------------------------	------------------------------

Since you have **7+ forms now**, you can start with **one DB and separate tables** — upgrade later if required.

🔗 Microservices + FastAPI — Example Structure

backend/

├─ lost_service/

| ├─ main.py

| ├─ database.py

| ├─ models.py

| └─ schemas.py

├─ order_received_service/

| ├─ main.py

| ├─ database.py

| ├─ models.py

| └─ schemas.py

```
└─ crm_service/
  │ └─ main.py
  │ └─ database.py
  │ └─ models.py
  │ └─ schemas.py
```

Each service:

- ✓ Runs on a different port
- ✓ Has different API endpoints
- ✓ Can be deployed individually

Your React frontend can connect to each:

Form Page	API Endpoint
-----------	--------------

LOST form	http://localhost:8001/lost
-----------	----------------------------

Order Received	http://localhost:8002/orders
----------------	------------------------------

CRM Leads	http://localhost:8003/crm-leads
-----------	---------------------------------

Communication between microservices

Later, if needed:

- **Kafka / RabbitMQ** for internal events
- **API Gateway (Kong / Traefik / NGINX)** for routing

Summary

Question	Answer
Can each form act independently?	✓ YES
Can each form have separate DB?	✓ YES (microservices recommended)
Is this scalable?	✓ Very scalable
Do you need it right now?	Optional — depends on project size

Next Step

If you want, I can create:

🔧 Full microservice starter structure:

- 1 backend per form
- PostgreSQL connection
- Docker support
- Deployment ready

Or

🧩 Unified API with multiple endpoints (simpler)

Just reply with one of these:

- ♦ **A — Build Microservice architecture**
- ♦ **B — One backend with all forms inside**

I'll then generate a **complete project folder structure + code** for you.

You said:

What are the dependencies I require for Python FastAPI and postgres

ChatGPT said:

To build a backend using **Python FastAPI + PostgreSQL**, you need a few essential dependencies for:

✓ **FastAPI framework**

✓ **Database connection (PostgreSQL)**

✓ **ORM layer (SQLAlchemy)**

✓ **Data validation (Pydantic)**

🔧 Required Dependencies

Purpose	Package
FastAPI framework	fastapi
Development server	uvicorn
ORM for DB operations	sqlalchemy
PostgreSQL driver	psycopg2-binary
Validation / serialization	pydantic
CORS for React / frontend fastapi[all] OR fastapi + fastapi-cors (optional)	

↓ Install core dependencies:

```
pip install fastapi uvicorn sqlalchemy psycpg2-binary pydantic
```

↓ Optional but recommended packages:

Need	Install
CORS for frontend	pip install fastapi[all] OR pip install starlette
Environment variable management	pip install python-dotenv
Alembic migration tool (optional)	pip install alembic

Example:

```
pip install python-dotenv alembic
```

🚀 requirements.txt example

You can create this file and run `pip install -r requirements.txt` later:

```
fastapi
uvicorn
sqlalchemy
psycpg2-binary
pydantic
python-dotenv
alembic
```

🚀 Minimal backend setup after installation

Run FastAPI server:

```
uvicorn main:app --reload
```

Check docs:

```
http://localhost:8000/docs
```

⚠️ Notes

Package	Important Notes
psycopg2-binary	Needed only for PostgreSQL, use binary instead of psycopg2 because it avoids compilation issues
SQLAlchemy	Works with both sync & async — we are using sync for simplicity
Uvicorn	Required to run FastAPI server

Summary

Component	Mandatory?
fastapi	✓ required
uvicorn	✓ required
sqlalchemy	✓ required
psycopg2-binary	✓ required
pydantic	✓ required
python-dotenv	Optional
alembic	Optional

If you want, I can provide next:

- ◆ A ready-made **backend zip** using these dependencies
- ◆ **Docker setup** for FastAPI + PostgreSQL
- ◆ **Multiple microservices structure**

Just type “**continue**” if you'd like that. 