Google Sheets to database microservice Shared by Areesh

This is a copy of a chat between Claude and Areesh. Content may include unverified or unsafe content that do not represent the views of Anthropic. Shared snapshot may contain attachments and data not displayed here. Report

A I want to break this down into very simple microservices. Eg - one would be to fetch data from google sheet that contains marketing tofu and stores in a db table. We'll only work on one microservice at a time. We'll keep the structure of the table the same as google sheets and add another column "UserID" that will basically be a 12 digit cleaned phone number (present in the g-sheet).
So in this instance we will only fetch data from G-sheets, store it in a DB of our choice (light and free) the do this everyday to fetch only new records

🧠 Ultra Think Mode: Activated

Let me analyze your problem space deeply before we dive into microservice design...

📊 Initial Analysis

I've reviewed your comprehensive documentation. You're building a **two-part funnel tracking system** where:

- **Part 1**: Marketing (TOFU) → Engagement (MOFU) → Lead Assignment
- **Part 2**: Sales conversions post-assignment

The core blocker: Google Sheets rate limits are killing your current workflow.

🎯 Microservice #1: TOFU Marketing Data Ingestion

You want to start with the **simplest, most foundational piece** - and that's smart. Let's architect this first microservice with absolute clarity.

Service Purpose

Fetch performance marketing leads from Google Sheets → Store in database → Run daily to capture only new records

---

🏗️ Architecture Blueprint for Microservice #1

Input Source

- **Google Sheet**: Auto-populated by Make.com automation
- **Columns** (9 fields):

- Name
- Email
- Phone number
- City
- Question 1
- utmSource
- utmMedium
- utmCamp
- created date

Transformation Logic

1. **Phone Normalization** (reuse from Streamlit):

python

```
# Extract last 10 digits only
digits = re.sub(r"\\D", "", phone_value)
if len(digits) >= 10:
    digits = digits[-10:]
```

2. **UserID Generation**:

python

```
# Format: "91" + 10-digit phone (zero-padded if needed)
user_id = f"91{digits[-10:].zfill(10)}"
```

3. **Data Validation**:
   - Drop rows with invalid phone numbers (not 10 digits after cleaning)
   - Keep all other fields as-is (no transformation)

Output Database Schema

Table Name: `marketing_leads_tofu`

Column	Type	Constraints	Notes
`id`	INTEGER	PRIMARY KEY, AUTO_INCREMENT	Internal DB ID
`name`	VARCHAR(255)		Raw from sheet
`email`	VARCHAR(255)		Raw from sheet

Column	Type	Constraints	Notes
`phone`	VARCHAR(20)		Raw from sheet (original)
`city`	VARCHAR(100)		Raw from sheet
`question_1`	TEXT		Raw from sheet
`utm_source`	VARCHAR(100)		Raw from sheet
`utm_medium`	VARCHAR(100)		Raw from sheet
`utm_camp`	VARCHAR(255)		Campaign name
`created_date`	DATETIME		Lead capture timestamp
`user_id`	VARCHAR(12)	INDEX	**Normalized: "91" + 10 digits**
`ingested_at`	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	When added to DB

Unique Constraint: Consider adding `UNIQUE(user_id, created_date)` to prevent exact duplicates

---

🔄 Daily Sync Logic: "New Records Only"

Critical Question I Need You To Answer:

How do we identify "new records"?

Option A: Timestamp-Based (Recommended)

- Track last successful sync timestamp
- Query Google Sheets where `created_date > last_sync_timestamp`
- **Pros**: Clean, efficient, predictable
- **Cons**: Requires Google Sheets to have proper timestamps

Option B: Row-Based

- Track last processed row number
- Fetch rows after that number
- **Pros**: Simple
- **Cons**: Fragile if rows are deleted/reordered

Option C: Comparison-Based

- Pull all data, compare against existing DB records
- Insert only missing `user_id + created_date` combinations
- **Pros**: Most robust, handles data corrections

- **Cons**: Higher API usage, more processing

My Recommendation: Option A if your Google Sheet has reliable `created_date` values. Otherwise **Option C** for robustness.

Which approach fits your current setup?

🛠️ Technology Stack Options

Database Choice (Lightweight + Free Tier)

Option 1: PostgreSQL (Supabase Free Tier) ⭐ RECOMMENDED

- 500MB database
- Unlimited API requests
- Built-in REST API
- Easy setup
- **Why**: Battle-tested, scales well, free tier generous

Option 2: SQLite (Local File)

- Zero cost
- No network overhead
- **Why**: Simplest possible, but requires file hosting

Option 3: MySQL (PlanetScale Free Tier)

- 5GB storage
- Good performance
- **Why**: Alternative to PostgreSQL

My Recommendation: **Supabase PostgreSQL** - best balance of simplicity, features, and future scalability.

⚙️ Execution Environment Options

Option 1: GitHub Actions (Free tier: 2,000 mins/month) ⭐ RECOMMENDED

- Scheduled workflows (cron)
- No server maintenance
- Easy to version control

- **Why**: Zero infra, perfect for daily batch jobs

Option 2: Python Script + Cron (Self-hosted)

- Run on any server/laptop
- Full control
- **Why**: Maximum flexibility

Option 3: Cloud Functions (Google Cloud/AWS Lambda)

- Pay-per-execution
- Serverless
- **Why**: Only if you want auto-scaling (overkill for this)

My Recommendation: GitHub Actions - zero maintenance, perfect for daily sync.

📝 Detailed Implementation Steps

Step 1: Setup Database (15 mins)

1. Create Supabase account
2. Create new project
3. Run SQL to create `marketing_leads_tofu` table
4. Save connection credentials

Step 2: Google Sheets API Setup (20 mins)

1. Enable Google Sheets API in GCP Console
2. Create Service Account
3. Share your Google Sheet with service account email
4. Download credentials JSON

Step 3: Write Python Script (2 hours)

Core components:

```python
# 1. Fetch data from Google Sheets (gspread library)
# 2. Normalize phone → generate UserID
# 3. Filter new records only
# 4. Insert into Supabase PostgreSQL (psycopg2)
# 5. Log success/failures
```

Step 4: GitHub Actions Workflow (30 mins)

yaml

```yaml
name: Daily TOFU Lead Sync
on:
  schedule:
    - cron: '0 2 * * *'  # Run at 2 AM IST daily
  workflow_dispatch:  # Manual trigger option

jobs:
  sync:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Python
      - name: Install dependencies
      - name: Run sync script
      - name: Log results
```

Step 5: Testing & Validation (1 hour)

- Test with sample data
- Verify deduplication logic
- Check error handling
- Validate UserID generation

🚨 Critical Questions Before We Proceed

1. Google Sheet Access

- Do you have the **exact Google Sheet URL/ID** for TOFU leads?
- Is the `created_date` column reliable and always populated?
- Approximate row count currently? (to estimate API limits)

2. New Records Definition

- Confirm: We use `created_date` to identify new records, correct?
- Should we handle **updates** to existing records, or only **inserts**?

3. Error Handling

- What happens if Google Sheets API fails? (Retry? Alert?)

- What happens if database write fails? (Queue? Skip?)

4. Data Volume

- How many new leads per day typically? (10? 100? 1000?)
- Any historical data to backfill initially?

5. Deduplication

- If same `user_id` appears twice with different `created_date`, do we:
  - Keep both records? (track re-engagement)
  - Keep only first? (first-touch attribution)
  - Update existing? (latest data wins)

📊 Success Metrics for Microservice #1

Before we build, let's define success:

✅ Functional Requirements

☐ Syncs daily at 2 AM IST automatically
☐ Fetches only new records (no duplicates)
☐ Generates valid UserID for all records
☐ Stores data in PostgreSQL table
☐ Logs sync results (records processed, errors)

✅ Non-Functional Requirements

☐ Sync completes in < 5 minutes
☐ Zero manual intervention needed
☐ Clear error notifications if sync fails
☐ Handles up to 1000 new leads per day

🎯 Next Steps - Your Input Needed

Before I create the detailed technical specification and code scaffolding, please answer:

1. **Confirm**: Is the Google Sheet structure exactly as documented (9 columns)?
2. **Decide**: New record detection - Option A (timestamp) or Option C (comparison)?
3. **Clarify**: Deduplication strategy for same UserID?
4. **Approve**: Supabase PostgreSQL + GitHub Actions stack?
5. **Provide**: Google Sheet ID/URL (or confirm you'll provide during setup)

Once you confirm these details, I'll create:

- ✅ Complete database schema SQL
- ✅ Python script with full logic
- ✅ GitHub Actions workflow YAML
- ✅ Step-by-step setup guide
- ✅ Testing checklist

Ready to proceed? Let me know your answers to the above questions! 🚀

A

Option A or B is fine with me.
Supabase works for me.
I want to use GitHub Actions, maybe we can explore Python Script + Cron later, let's go with GitHub Actions for now.
1. Google Sheet Access - I have exact Google Sheet URL/ID for TOFU leads (these will be two sheets), `created_date` column reliable and always populated, around 100000-200000 leads monthly.

Show more

🚀 Perfect! Let's Build Microservice #1

Based on your answers, I'm now creating a **complete, production-ready blueprint**. Let me think through the architecture considering your scale (100-200K leads/month, 400-500K historical).

📋 Final Architecture Decisions

✅ **Sync Strategy**: Option A (Timestamp-based) - Most efficient for your volume
✅ **Database**: Supabase PostgreSQL
✅ **Execution**: GitHub Actions (daily cron)
✅ **Deduplication**: Keep all records (track re-engagement)
✅ **Initial Load**: Separate one-time backfill process for 400-500K historical records

---

🗄️ Database Schema (Supabase PostgreSQL)

Table: `marketing_leads_tofu`

sql

```
-- Create the main table
CREATE TABLE marketing_leads_tofu (
    id BIGSERIAL PRIMARY KEY,

    -- Raw data from Google Sheets (exactly as-is)
    name VARCHAR(255),
    email VARCHAR(255),
    phone VARCHAR(50),  -- Store original format
    city VARCHAR(100),
    question_1 TEXT,
    utm_source VARCHAR(100),
    utm_medium VARCHAR(100),
    utm_camp VARCHAR(255),
    created_date TIMESTAMP NOT NULL,

    -- Computed fields
    user_id VARCHAR(12) NOT NULL,  -- Normalized: "91" + 10 digits

    -- Metadata
    sheet_source VARCHAR(50) NOT NULL,  -- Which sheet this came from
    ingested_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Indexes for performance
    INDEX idx_user_id (user_id),
    INDEX idx_created_date (created_date),
    INDEX idx_sheet_source (sheet_source),
    INDEX idx_ingested_at (ingested_at)
);

-- Composite index for common queries
CREATE INDEX idx_user_created ON marketing_leads_tofu(user_id, created_date);

-- Table to track sync state
CREATE TABLE sync_state (
    id SERIAL PRIMARY KEY,
    sheet_name VARCHAR(50) UNIQUE NOT NULL,
    last_synced_date TIMESTAMP,
    last_sync_status VARCHAR(20),  -- 'success' or 'failed'
    records_processed INTEGER DEFAULT 0,
    errors TEXT,
    synced_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert initial state for your two sheets
INSERT INTO sync_state (sheet_name, last_synced_date, last_sync_status)
```

~~~
VALUES
    ('sheet_1', '2020-01-01 00:00:00', 'pending'),
    ('sheet_2', '2020-01-01 00:00:00', 'pending');
```


**Storage Estimate:**
- 500K rows × ~500 bytes/row = ~250MB (well within Supabase free tier of 500MB)

---

## 🐍 Python Script Architecture

### **File Structure**
```

tofu-lead-sync/
├── .github/
│   └── workflows/
│       ├── daily-sync.yml          # Daily incremental sync
│       └── historical-backfill.yml # One-time backfill (manual trigger)
├── src/
│   ├── sync_leads.py               # Main sync logic
│   ├── backfill_historical.py     # Historical data loader
│   ├── config.py                   # Configuration
│   └── utils.py                    # Shared utilities
├── requirements.txt
├── .env.example
└── README.md
~~~

python

```python
import os
from dotenv import load_dotenv

load_dotenv()

# Google Sheets Configuration
GOOGLE_SHEETS = {
    'sheet_1': {
        'id': os.getenv('SHEET_1_ID'),
        'name': os.getenv('SHEET_1_NAME', 'Sheet1'),
        'identifier': 'sheet_1'
    },
    'sheet_2': {
        'id': os.getenv('SHEET_2_ID'),
        'name': os.getenv('SHEET_2_NAME', 'Sheet1'),
        'identifier': 'sheet_2'
    }
}

# Supabase PostgreSQL Configuration
DB_CONFIG = {
    'host': os.getenv('SUPABASE_HOST'),
    'database': os.getenv('SUPABASE_DB'),
    'user': os.getenv('SUPABASE_USER'),
    'password': os.getenv('SUPABASE_PASSWORD'),
    'port': os.getenv('SUPABASE_PORT', '5432')
}

# Google Service Account JSON (base64 encoded in GitHub Secrets)
GOOGLE_SERVICE_ACCOUNT_JSON = os.getenv('GOOGLE_SERVICE_ACCOUNT_JSON')

# Expected Google Sheets columns (in order)
EXPECTED_COLUMNS = [
    'Name',
    'Email',
    'Phone number',
    'City',
    'Question 1',
    'utmSource',
    'utmMedium',
    'utmCamp',
    'created date'
]
```

---

3. `src/utils.py`

python

```python
import re
import logging
from datetime import datetime

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)


def normalize_phone(phone_value):
    """
    Normalize phone number to 10 digits.
    Returns empty string if invalid.
    """
    if not phone_value:
        return ""

    # Remove all non-digit characters
    digits = re.sub(r"\\D", "", str(phone_value))

    # Take last 10 digits
    if len(digits) >= 10:
        digits = digits[-10:]

    # Validate exactly 10 digits
    if len(digits) != 10:
        return ""

    return digits


def generate_user_id(phone):
    """
    Generate UserID: "91" + 10-digit phone (zero-padded).
    """
    normalized = normalize_phone(phone)
    if not normalized:
        return ""

    return f"91{normalized.zfill(10)}"


def parse_created_date(date_value):
    """
    Parse created_date to datetime object.
    Handles multiple date formats from Google Sheets.
    """
    if not date_value:
        return None

    # Try different formats
    formats = [
        '%d/%m/%Y %H:%M:%S',
        '%d/%m/%Y',
        '%Y-%m-%d %H:%M:%S',
        '%Y-%m-%d',
        '%m/%d/%Y %H:%M:%S',
        '%m/%d/%Y'
    ]

    for fmt in formats:
        try:
            return datetime.strptime(str(date_value), fmt)
        except ValueError:
            continue

    logger.warning(f"Could not parse date: {date_value}")
    return None


def validate_row(row, expected_columns):
    """
    Validate that row has all expected columns and required fields.
    Returns (is_valid, error_message).
    """
    if len(row) < len(expected_columns):
        return False, f"Row has only {len(row)} columns, expected {len(expected_

    # Check required fields
    phone = row[2]  # Phone number column
    created_date = row[8]  # created date column

    if not phone:
        return False, "Missing phone number"

    if not created_date:
        return False, "Missing created_date"

    # Validate phone can be normalized
    if not normalize_phone(phone):
        return False, f"Invalid phone number: {phone}"

    # Validate date can be parsed
    if not parse_created_date(created_date):
        return False, f"Invalid created_date: {created_date}"

    return True, None
```

Start your own conversation