Week 5 Lab: Git & API Integration

CS 203: Software Tools and Techniques for AI

Duration: 3 hours

Lab Overview

By the end of this lab, you will:

- Master Git workflow (init, add, commit, branch, merge)
- Collaborate using GitHub
- Call external APIs from Python
- Integrate Gemini API with FastAPI
- Build a complete Text Tools API

Structure:

- Part 1: Git Basics (45 min)
- Part 2: Git Collaboration (30 min)
- Part 3: External API Calls (45 min)
- Part 4: Build Text Tools API (60 min)

Setup (10 minutes)

Install tools:

```
# Verify Git installation
git --version

# Install Python packages
pip install fastapi uvicorn requests httpx python-dotenv google-genai

# Configure Git (if not done)
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Get Gemini API Key:

1. Visit https://aistudio.google.com/apikey
2. Create API key
3. Save for later use

Exercise 1.1: Git Initialization (10 min)

Task: Create and initialize a Git repository

```
# Create project directory
mkdir text-tools-api
cd text-tools-api

# Initialize Git
git init

# Check status
git status

# Create initial files
echo "# Text Tools API" > README.md
echo "*.pyc" > .gitignore
echo "__pycache__/" >> .gitignore
echo ".env" >> .gitignore

# Stage and commit
git add .
git commit -m "Initial commit: Add README and gitignore"
```

Exercise 1.2: Basic Git Workflow (15 min)

Task: Create and track files

Create `main.py`:

```
from fastapi import FastAPI

app = FastAPI(title="Text Tools API")

@app.get("/")
def read_root():
    return {"message": "Text Tools API v1.0"}
```

Git workflow:

```
# Check status
git status

# Stage file
git add main.py
```

Exercise 1.3: Making Changes (10 min)

Task: Modify code and track changes

Add to `main.py`:

```
@app.get("/health")
def health_check():
    return {"status": "healthy"}
```

Track changes:

```
# See what changed
git diff

# Stage and commit
git add main.py
git commit -m "Add health check endpoint"

# View detailed history
git log
```

Exercise 1.4: Branching (10 min)

Task: Create and work on a feature branch

```
# Create and switch to new branch
git checkout -b feature/sentiment-analysis

# Verify current branch
git branch

# Add new code
```

Add to `main.py`:

```
from pydantic import BaseModel

class TextInput(BaseModel):
    text: str

@app.post("/sentiment")
def analyze_sentiment(input: TextInput):
    return {"text": input.text, "sentiment": "positive"}
```

Exercise 1.5: Merging (10 min)

Task: Merge feature branch into main

```
# Switch to main branch
git checkout main

# Merge feature branch
git merge feature/sentiment-analysis

# View history
git log --oneline --graph

# Delete feature branch (optional)
git branch -d feature/sentiment-analysis

# List branches
git branch
```

Exercise 2.1: GitHub Setup (10 min)

Task: Create GitHub repository and push

1. Go to https://github.com
2. Click "New repository"
3. Name it "text-tools-api"
4. Don't initialize with README (we have one)
5. Create repository

Connect and push:

```
# Add remote
git remote add origin https://github.com/YOUR_USERNAME/text-tools-api.git

# Push
git push -u origin main

# Verify
git remote -v
```

Exercise 2.2: Collaboration Workflow (20 min)

Task: Practice branch-push-PR workflow

```
# Create feature branch
git checkout -b feature/summarize

# Add summarization endpoint
```

Add to `main.py`:

```
@app.post("/summarize")
def summarize_text(input: TextInput):
    return {
        "text": input.text,
        "summary": "Summary placeholder"
    }
```

```
# Commit
git add main.py
```

Exercise 2.3: Pull Latest Changes (10 min)

Task: Sync local repository

```
# Switch to main
git checkout main

# Pull latest changes
git pull origin main

# Verify changes
git log --oneline

# Clean up local feature branch
git branch -d feature/summarize

# View all branches (including remote)
git branch -a
```

Exercise 3.1: Calling Public APIs (15 min)

Task: Fetch data from JSONPlaceholder

Create `test_api.py`:

```
import requests

# GET request
response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
print("Status:", response.status_code)
print("Data:", response.json())

# POST request
new_post = {
    "title": "My Post",
    "body": "This is the content",
    "userId": 1
}

response = requests.post(
    "https://jsonplaceholder.typicode.com/posts",
    json=new_post
```

Exercise 3.2: Error Handling (15 min)

Task: Add robust error handling

Update `test_api.py`:

```
import requests

def fetch_post(post_id):
    try:
        response = requests.get(
            f"https://jsonplaceholder.typicode.com/posts/{post_id}",
            timeout=5
        )
        response.raise_for_status()
        return response.json()
    except requests.exceptions.Timeout:
        return {"error": "Request timed out"}
    except requests.exceptions.HTTPError as e:
        return {"error": f"HTTP error: {e}"}
    except requests.exceptions.RequestException as e:
        return {"error": f"Error: {e}"}

# Test with valid and invalid IDs
```

Exercise 3.3: Rate Limiting (15 min)

Task: Implement rate-limited API calls

Create `rate_limit_test.py`:

```
import requests
import time

def fetch_with_rate_limit(urls, calls_per_second=2):
    results = []
    delay = 1 / calls_per_second

    for url in urls:
        start = time.time()
        response = requests.get(url)
        results.append(response.json())

        elapsed = time.time() - start
        if elapsed < delay:
            time.sleep(delay - elapsed)

    return results

urls = [f"https://jsonplaceholder.typicode.com/posts/{i}" for i in range(1, 6)]
posts = fetch_with_rate_limit(urls, calls_per_second=2)
```

Exercise 3.4: Async API Calls (15 min)

Task: Fetch multiple URLs concurrently

Create `async_api.py`:

```
import asyncio
import httpx

async def fetch_post(client, post_id):
    url = f"https://jsonplaceholder.typicode.com/posts/{post_id}"
    response = await client.get(url)
    return response.json()

async def fetch_all_posts(post_ids):
    async with httpx.AsyncClient() as client:
        tasks = [fetch_post(client, pid) for pid in post_ids]
        return await asyncio.gather(*tasks)

# Fetch posts 1-10 concurrently
post_ids = range(1, 11)
posts = asyncio.run(fetch_all_posts(post_ids))
print(f"Fetched {len(posts)} posts")
```

Exercise 4.1: Setup Environment (10 min)

Task: Configure API keys securely

Create `.env` file:

```
GEMINI_API_KEY=your_gemini_api_key_here
```

Update `main.py`:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from google import genai
import os
from dotenv import load_dotenv

load_dotenv()

app = FastAPI(title="Text Tools API", version="1.0")

# Initialize Gemini client
client = genai.Client(api_key=os.getenv("GEMINI_API_KEY"))
```

Exercise 4.2: Sentiment Analysis (15 min)

Task: Implement real sentiment analysis

Add to `main.py`:

```
@app.post("/sentiment")
def analyze_sentiment(input: TextInput):
    if not input.text.strip():
        raise HTTPException(status_code=400, detail="Text cannot be empty")

    try:
        prompt = f"Analyze sentiment (Positive/Negative/Neutral): {input.text}"

        response = client.models.generate_content(
            model="models/gemini-2.0-flash-exp",
            contents=prompt
        )

        return {
            "text": input.text,
            "sentiment": response.text.strip()
        }
```

Exercise 4.3: Text Summarization (15 min)

Task: Implement text summarization

Add to `main.py`:

```
class SummarizeRequest(BaseModel):
    text: str
    max_sentences: int = 3

@app.post("/summarize")
def summarize_text(input: SummarizeRequest):
    if not input.text.strip():
        raise HTTPException(status_code=400, detail="Text cannot be empty")

    try:
        prompt = f"""
        Summarize the following text in {input.max_sentences} sentences:

        {input.text}
        """

        response = client.models.generate_content(
            model="models/gemini-2.0-flash-exp",
            contents=prompt
        )

        return {
            "original_length": len(input.text),
            "summary": response.text.strip(),
            "compression_ratio": round(len(response.text) / len(input.text), 2)
        }
```

Exercise 4.4: Entity Extraction (15 min)

Task: Extract named entities

Add to `main.py`:

```
import json

@app.post("/extract-entities")
def extract_entities(input: TextInput):
    if not input.text.strip():
        raise HTTPException(status_code=400, detail="Text cannot be empty")

    try:
        prompt = f"""
        Extract entities as JSON:
        {{"Person": [], "Organization": [], "Location": [], "Date": []}}

        Text: {input.text}
        """

        response = client.models.generate_content(
            model="models/gemini-2.0-flash-exp",
            contents=prompt
        )

        entities = json.loads(response.text)
        return {"text": input.text, "entities": entities}
    except json.JSONDecodeError:
        raise HTTPException(status_code=500, detail="Failed to parse entities")
```

Exercise 4.5: Translation (15 min)

Task: Add translation endpoint

Add to `main.py`:

```
class TranslationRequest(BaseModel):
    text: str
    target_language: str

@app.post("/translate")
def translate_text(input: TranslationRequest):
    if not input.text.strip():
        raise HTTPException(status_code=400, detail="Text cannot be empty")

    try:
        prompt = f"Translate to {input.target_language}: {input.text}"

        response = client.models.generate_content(
            model="models/gemini-2.0-flash-exp",
            contents=prompt
        )

        return {
            "original": input.text,
            "translated": response.text.strip(),
            "target_language": input.target_language
        }
```

Exercise 4.6: Question Answering (15 min)

Task: Add QA endpoint

Add to `main.py`:

```
class QARequest(BaseModel):
    context: str
    question: str

@app.post("/qa")
def answer_question(input: QARequest):
    if not input.context.strip() or not input.question.strip():
        raise HTTPException(status_code=400, detail="Context and question required")

    try:
        prompt = f"""
        Context: {input.context}

        Question: {input.question}

        Provide a concise answer based only on the context.
        """

        response = client.models.generate_content(
            model="models/gemini-2.0-flash-exp",
            contents=prompt
        )

        return {
            "question": input.question,
            "answer": response.text.strip()
        }
```

Exercise 4.7: Add Usage Tracking (10 min)

Task: Log API usage

Add to `main.py`:

```
import logging
from datetime import datetime

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Add to each endpoint
@app.post("/sentiment")
def analyze_sentiment(input: TextInput):
    logger.info(f"[{datetime.now()}] Sentiment analysis - Text length: {len(input.text)}")
    # ... rest of the code
```

Add usage stats endpoint:

```
from collections import defaultdict
```

Exercise 4.8: Test Your API (10 min)

Task: Test all endpoints

```
# Root
http http://localhost:8000/

# Health
http http://localhost:8000/health

# Sentiment
http POST http://localhost:8000/sentiment text="This is amazing!"

# Summarize
http POST http://localhost:8000/summarize \
  text="Long text here ..." max_sentences=2

# Extract entities
http POST http://localhost:8000/extract-entities \
  text="Apple CEO Tim Cook met PM Modi in Delhi on Monday."

# Translate
http POST http://localhost:8000/translate \
  text="Hello, world!" target_language="Spanish"

# QA
http POST http://localhost:8000/qa \
  context="The sky is blue" question="What color is the sky?"
```

Exercise 4.9: Commit Your Work (10 min)

Task: Save progress to Git

```
# Check status
git status

# Stage files
git add .

# Commit
git commit -m "Add complete Text Tools API with LLM integration

- Sentiment analysis
- Text summarization
- Entity extraction
- Translation
- Question answering
- Usage tracking
- Environment configuration"

# Push to GitHub
git push
```

Exercise 5: Add Documentation (15 min)

Task: Update README

Update `README.md`:

```
# Text Tools API

AI-powered text processing API built with FastAPI and Gemini.

## Features

- Sentiment Analysis
- Text Summarization
- Entity Extraction
- Translation
- Question Answering

## Setup

1. Clone repository
2. Install dependencies: `pip install -r requirements.txt`
3. Create `.env` with `GEMINI_API_KEY=your_key`
4. Run: `uvicorn main:app --reload`

## Endpoints

- POST `/sentiment` - Analyze sentiment
- POST `/summarize` - Summarize text
- POST `/extract-entities` - Extract named entities
- POST `/translate` - Translate text
- POST `/qa` - Answer questions
```

Exercise 6: Create requirements.txt (10 min)

Task: Document dependencies

Create `requirements.txt`:

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
python-dotenv==1.0.0
google-genai==0.2.0
requests==2.31.0
httpx==0.25.0
```

Commit:

```
git add README.md requirements.txt
git commit -m "Add documentation and requirements file"
git push
```

Bonus Exercise 1: Add Caching (15 min)

Task: Cache LLM responses

Add to `main.py`:

```
from functools import lru_cache

@lru_cache(maxsize=100)
def get_sentiment(text: str) → str:
    prompt = f"Analyze sentiment (Positive/Negative/Neutral): {text}"
    response = client.models.generate_content(
        model="models/gemini-2.0-flash-exp",
        contents=prompt
    )
    return response.text.strip()

@app.post("/sentiment")
def analyze_sentiment(input: TextInput):
    try:
        sentiment = get_sentiment(input.text)
        return {"text": input.text, "sentiment": sentiment}
    except Exception as e:
```

Bonus Exercise 2: Batch Processing (15 min)

Task: Process multiple texts at once

Add to `main.py`:

```
from typing import List

class BatchTextInput(BaseModel):
    texts: List[str]

@app.post("/sentiment/batch")
def batch_sentiment(input: BatchTextInput):
    results = []

    for text in input.texts:
        try:
            sentiment = get_sentiment(text)
            results.append({
                "text": text,
                "sentiment": sentiment
            })
        except Exception as e:
            results.append({
                "text": text,
                "error": str(e)
            })
```

Bonus Exercise 3: Rate Limiting (15 min)

Task: Add rate limiting to protect API

```
pip install slowapi
```

Add to `main.py`:

```
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address
from slowapi.errors import RateLimitExceeded

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

@app.post("/sentiment")
@limiter.limit("10/minute")
def analyze_sentiment(input: TextInput, request: Request):
    # ... existing code
```

Bonus Exercise 4: Docker Deployment (20 min)

Task: Containerize your API

Create `Dockerfile`:

```
FROM python:3.11-slim

WORKDIR /app

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

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Build and run:

```
docker build -t text-tools-api .
docker run -p 8000:8000 --env-file .env text-tools-api
```

Deliverables

Submit the following:

1. GitHub repository URL
2. `main.py` - Complete API implementation
3. `README.md` - Documentation
4. `requirements.txt` - Dependencies
5. `.gitignore` - Ignored files (but NOT .env file itself)
6. Git history showing:
    - Multiple commits with clear messages
    - At least one feature branch and merge
    - No sensitive data committed

Bonus:

- Caching implementation

Testing Checklist

Before submission, verify:

- [ ] All endpoints work correctly
- [ ] Error handling for invalid input
- [ ] Environment variables used (no hardcoded keys)
- [ ] API documentation accessible at `/docs`
- [ ] Git history is clean and organized
- [ ] .env NOT committed to Git
- [ ] README has clear instructions
- [ ] requirements.txt is complete
- [ ] Code is properly formatted

Common Issues and Solutions

Issue: API key not found

- Check `.env` file exists
- Verify `python-dotenv` installed
- Use `load_dotenv()` at top of file

Issue: Git merge conflicts

- Use `git status` to see conflicts
- Edit files to resolve
- `git add` and `git commit`

Issue: LLM API errors

- Check API key is valid
- Verify internet connection
- Check API quotas/limits

Git Commands Summary

```
git init                      # Initialize repo
git status                    # Check status
git add <file>               # Stage file
git commit -m "message"      # Commit changes
git log --oneline            # View history
git branch <name>            # Create branch
git checkout <branch>        # Switch branch
git merge <branch>           # Merge branch
git remote add origin <url> # Add remote
git push -u origin main      # Push to remote
git pull                     # Pull changes
```

Resources

Documentation:

- FastAPI: https://fastapi.tiangolo.com/
- Gemini API: https://ai.google.dev/
- Git: https://git-scm.com/doc
- GitHub: https://docs.github.com/

Testing APIs:

- Your API docs: http://localhost:8000/docs
- JSONPlaceholder: https://jsonplaceholder.typicode.com/

Tools:

- httpie: https://httpie.io/
- Postman: https://www.postman.com/

Excellent Work!

Next week: Active Learning

Questions? Office hours: Tomorrow 3-5 PM