Week 5: Git & API Integration

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

This Week's Journey

Part 1: Git Fundamentals

- Version control concepts and Git internals
- Basic Git workflow with visual diagrams
- Branching and merging strategies
- Collaboration with GitHub

Part 2: Calling External APIs

- Using `requests` and `httpx`
- Error handling and retries
- Rate limiting and pagination

Part 3: Integrating LLM APIs with FastAPI

Why Version Control?

The Problem:

- `project_final.py`
- `project_final_v2.py`
- `project_final_ACTUALLY_FINAL.py`
- `project_final_this_time_i_mean_it.py`

The Solution: Git

- Track every change
- Go back to any version
- Collaborate without conflicts
- Experiment safely with branches

What is Git?

Distributed Version Control System

Key Concepts:

- **Repository (repo)**: Project folder tracked by Git
- **Commit**: Snapshot of your project at a point in time
- **Branch**: Parallel version of your code
- **Remote**: Server copy (e.g., GitHub, GitLab)

Git vs GitHub:

- Git: Version control system (tool)
- GitHub: Hosting service for Git repositories (platform)

Git Internals: The `.git` Directory

When you run `git init`, Git creates a `.git/` directory:

```
.git/
├── HEAD              # Points to current branch
├── config            # Repository configuration
├── objects/          # Database of all content
│   ├── 2e/           # First 2 chars of SHA-1 hash
│   │   └── 9f3a... # Rest of hash (file content)
│   └── ...
├── refs/             # Pointers to commits
│   ├── heads/        # Local branches
│   │   └── main
│   └── remotes/      # Remote branches
│       └── origin/
│           └── main
└── index             # Staging area
```

Git stores everything as content-addressed objects!

Installing and Configuring Git

Install:

```
# macOS
brew install git

# Ubuntu/Debian
sudo apt install git

# Verify
git --version
```

Configure:

```
# Set your identity
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

# Check configuration
```

Creating Your First Repository

Initialize a new repo:

```
# Create project directory
mkdir my-project
cd my-project

# Initialize Git
git init

# Check status
git status
```

Output:

```
Initialized empty Git repository in /path/to/my-project/.git/
On branch master
No commits yet
```

Git Objects: The Building Blocks

Git stores four types of objects (all in `.git/objects/`):

1. **Blob** (Binary Large Object)
   - Stores file content
   - Identified by SHA-1 hash of content
2. **Tree**
   - Represents directory structure
   - Points to blobs and other trees
3. **Commit**
   - Points to a tree (snapshot)
   - Contains metadata (author, message, parent)

Git Object Model Visualized

```
graph TD
    C1[Commit a1b2c3] --> T1[Tree def456]
    C1 --> P1[Parent: null]
    C1 --> M1[Message: 'Initial commit']

    T1 --> B1[Blob: README.md]
    T1 --> B2[Blob: main.py]
    T1 --> T2[Tree: src/]

    T2 --> B3[Blob: utils.py]

    style C1 fill:#e8f5e9
    style T1 fill:#fff4e1
    style T2 fill:#fff4e1
    style B1 fill:#e1f5ff
    style B2 fill:#e1f5ff
    style B3 fill:#e1f5ff
```

Each commit is a snapshot of the entire project tree

SHA-1 Hashing: Git's Fingerprints

Git uses SHA-1 hashing to identify every object:

$$\text{SHA-1}(\text{content}) \rightarrow \text{40-character hexadecimal hash}$$

Example:

```
echo "Hello World" | git hash-object --stdin
# Output: 557db03de997c86a4a028e1ebd3a1ceb225be238
```

Properties:

- Same content → Same hash (deterministic)
- Different content → Different hash (collision-resistant)
- Hash serves as unique identifier

Git abbreviates hashes (first 7 chars usually sufficient):

The Three States

Git has three main states for files:

1. **Working Directory**: Where you edit files
2. **Staging Area (Index)**: Files ready to commit
3. **Repository (.git)**: Committed snapshots

Visual Flow:

```
graph LR
    A[Working Directory] -->|git add| B[Staging Area]
    B -->|git commit| C[Repository]
    C -->|git checkout| A

    style A fill:#e1f5ff
    style B fill:#fff4e1
    style C fill:#e8f5e9
```

Basic Git Workflow

1. Create or modify files:

```
echo "# My Project" > README.md
echo "print('Hello')" > main.py
```

2. Check status:

```
git status
```

3. Stage files:

```
git add README.md main.py
# Or add all: git add .
```

4. Commit:

Viewing History

See commit history:

```
# Full log
git log

# Condensed view
git log --oneline

# With graph
git log --oneline --graph --all

# Last N commits
git log -n 5
```

Example output:

```
a1b2c3d Add user authentication
e4f5g6h Fix validation bug
```

Understanding Commits

Each commit has:

- Unique hash (SHA-1): `a1b2c3d4e5f6...`
- Author and timestamp
- Commit message
- Parent commit(s)
- Snapshot of all files

Commit Anatomy:

```
git cat-file -p a1b2c3
# tree def456789abc
# parent e4f5g6h7i8j9  (if not first commit)
# author Alice <alice@example.com> 1234567890 +0000
# committer Alice <alice@example.com> 1234567890 +0000
#
```

Commit History as a Directed Acyclic Graph (DAG)

```
gitGraph
    commit id: "Initial commit"
    commit id: "Add user model"
    commit id: "Add validation"
    commit id: "Fix bug"
    commit id: "Add tests"
```

Each commit points to its parent(s):

```
graph RL
    C3[Commit C3] --> C2[Commit C2]
    C2 --> C1[Commit C1]
    C1 --> C0[Commit C0]

    C3 -.-> T3[Tree: snapshot 3]
    C2 -.-> T2[Tree: snapshot 2]
    C1 -.-> T1[Tree: snapshot 1]
    C0 -.-> T0[Tree: snapshot 0]
```

Good Commit Messages

Good commit messages:

```
# ✅ Good - Imperative mood, describes what the commit does
git commit -m "Add email validation to user registration"
git commit -m "Fix null pointer exception in login handler"
git commit -m "Refactor database connection pooling"

# ❌ Bad - Vague, not descriptive
git commit -m "fixed stuff"
git commit -m "asdf"
git commit -m "updates"
git commit -m "changes"
```

Conventional Commits Pattern:

```
git commit -m "feat: add user profile picture upload"
git commit -m "fix: resolve race condition in auth"
git commit -m "docs: update API documentation"
```

Viewing Changes

See what changed:

```
# Unstaged changes
git diff

# Staged changes
git diff --staged

# Changes in specific file
git diff main.py

# Compare commits
git diff a1b2c3d e4f5g6h
```

Ignoring Files

Create `.gitignore`:

```
# Python
__pycache__/
*.pyc
*.pyo
.env
venv/
.venv/

# IDE
.vscode/
.idea/
*.swp

# OS
.DS_Store
Thumbs.db

# Project-specific
```

Undoing Changes

Discard unstaged changes:

```
# Single file
git checkout -- main.py

# All files
git checkout -- .
```

Unstage files:

```
git reset HEAD main.py
```

Amend last commit:

```
# Fix commit message or add forgotten files
git add forgotten_file.py
git commit --amend -m "Updated commit message"
```

What is a Branch? (Conceptually)

A branch is just a pointer to a commit.

```
graph RL
    HEAD[HEAD] -.->|points to| MAIN
    MAIN[main] --> C3[Commit C3]
    C3 --> C2[Commit C2]
    C2 --> C1[Commit C1]

    style HEAD fill:#ffe1f5
    style MAIN fill:#e8f5e9
    style C3 fill:#e1f5ff
    style C2 fill:#e1f5ff
    style C1 fill:#e1f5ff
```

Key concepts:

- **Branch**: Movable pointer to a commit (stored in `.git/refs/heads/`)
- **HEAD**: Pointer to the current branch (stored in `.git/HEAD`)

Branching - Why?

Branches allow parallel development:

- **main/master**: Production-ready code
- **develop**: Integration branch
- **feature/user-auth**: New feature
- **bugfix/login-error**: Bug fix
- **experiment/new-algorithm**: Experimentation

Benefits:

- Work on features independently
- Don't break main code
- Easy experimentation

Creating and Switching Branches

Create branch:

```
git branch feature-login
```

Switch to branch:

```
git checkout feature-login
# Or in one command:
git checkout -b feature-login
```

List branches:

```
git branch
# * indicates current branch
```

Modern syntax (Git 2.23+):

Creating a Branch: Visual Step-by-Step

Initial state:

```
graph RL
    HEAD[HEAD] -.-> MAIN[main]
    MAIN --> C2[C2: Add tests]
    C2 --> C1[C1: Add model]

    style HEAD fill:#ffe1f5
    style MAIN fill:#e8f5e9
```

After `git branch feature`:

```
graph RL
    HEAD[HEAD] -.-> MAIN[main]
    MAIN --> C2[C2: Add tests]
    FEATURE[feature] --> C2
    C2 --> C1[C1: Add model]
```

Making Commits on a Branch

After `git commit -m "Add API integration"` on feature branch:

```
graph RL
    HEAD[HEAD] -.-> FEATURE[feature]
    FEATURE --> C3[C3: Add API]
    C3 --> C2[C2: Add tests]
    MAIN[main] --> C2
    C2 --> C1[C1: Add model]

    style HEAD fill:#ffe1f5
    style FEATURE fill:#e8f5e9
    style C3 fill:#c8e6c9
    style MAIN fill:#fff4e1
```

Notice:

- `feature` moved forward to C3
- `main` still points to C2

Working with Branches

Example workflow:

```
# Create and switch to new branch
git checkout -b feature-api-integration

# Make changes
echo "def call_api(): pass" >> api.py
git add api.py
git commit -m "Add API integration module"

# Switch back to main
git checkout main

# View all branches
git branch -a
```

Merging Branches

Merge feature into main:

```
# Switch to target branch
git checkout main

# Merge feature branch
git merge feature-login

# Delete merged branch (optional)
git branch -d feature-login
```

Types of merges:

- **Fast-forward**: Linear history (no divergence)
- **Three-way merge**: Creates merge commit (diverged branches)

Fast-Forward Merge

Scenario: `main` hasn't changed since `feature` was created.

Before merge:

```
graph RL
    HEAD[HEAD] -.-> MAIN[main]
    MAIN --> C2[C2]
    FEATURE[feature] --> C4[C4]
    C4 --> C3[C3]
    C3 --> C2
    C2 --> C1[C1]

    style MAIN fill:#fff4e1
    style FEATURE fill:#e8f5e9
```

After `git merge feature`:

```
graph RL
```

Three-Way Merge

Scenario: Both `main` and `feature` have new commits.

Before merge:

```
graph RL
    MAIN[main] --> C3[C3: main work]
    FEATURE[feature] --> C4[C4: feature work]
    C3 --> C2[C2: common ancestor]
    C4 --> C2
    C2 --> C1[C1]

    style MAIN fill:#fff4e1
    style FEATURE fill:#e8f5e9
    style C2 fill:#ffe1f5
```

After `git merge feature`:

```
graph RL
```

How Three-Way Merge Works

Git uses **three snapshots** to merge:

1. **Common ancestor** (C2): Where branches diverged
2. **Target branch** (C3): Current branch (`main`)
3. **Source branch** (C4): Branch being merged (`feature`)

Algorithm:

- If file changed in `feature` but not in `main` → take `feature` version
- If file changed in `main` but not in `feature` → take `main` version
- If file changed in both the same way → automatic merge
- If file changed in both differently → **CONFLICT**

Handling Merge Conflicts

Conflict occurs when:

- Same line edited in both branches
- File deleted in one, modified in other

Conflict markers in file:

```
<<<<<<< HEAD (Current Change - main branch)
print("Hello from main")
=======
print("Hello from feature")
>>>>>>> feature-branch (Incoming Change)
```

Resolving:

```
git merge feature-branch
# Auto-merging main.py
```

Conflict Resolution Visualized

```
sequenceDiagram
    participant User
    participant Git
    participant File

    User->>Git: git merge feature
    Git->>File: Check for conflicts
    File-->>Git: Conflict detected!
    Git-->>User: CONFLICT in main.py

    User->>File: Edit file, remove markers
    File-->>User: Resolved version

    User->>Git: git add main.py
    User->>Git: git commit -m "Resolve conflict"
    Git-->>User: Merge complete!

    Note over Git: Creates merge commit<br/>with two parents
```

Common Merge Conflict Example

Branch `main`:

```
def calculate_total(items):
    return sum(item.price for item in items)
```

Branch `feature`:

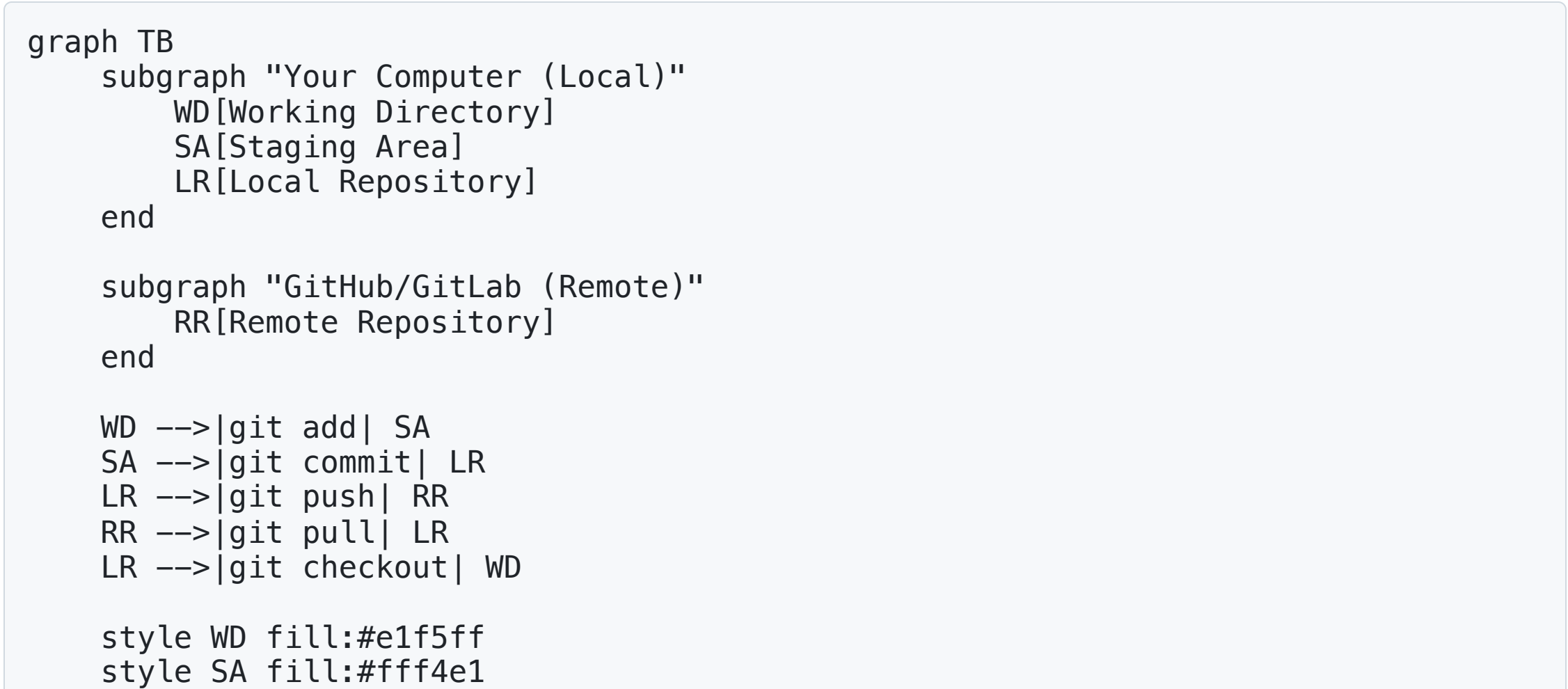
```
def calculate_total(items):
    return sum(item.price * item.quantity for item in items)
```

After `git merge feature` (conflict!):

```
def calculate_total(items):
<<<<<<< HEAD
    return sum(item.price for item in items)
=======
    return sum(item.price * item.quantity for item in items)
```

Remote Repositories

Local vs Remote:



Remote Branches

Remote-tracking branches are references to the state of remote branches:

```
.git/refs/
├── heads/                  # Local branches
│   ├── main
│   └── feature
└── remotes/                # Remote-tracking branches
    └── origin/
        ├── main
        └── feature
```

Naming convention: `remote-name/branch-name`

- `origin/main` : Remote branch `main` on `origin`
- `origin/feature` : Remote branch `feature` on `origin`

These are read-only snapshots of remote state!

Clone, Fetch, Pull, Push Visualized

`git clone` : Copy entire repository

```
sequenceDiagram
    participant Local
    participant Remote

    Remote->>Local: Clone entire history
    Note over Local: Creates local repo<br/>+ working directory<br/>+ origin remote
```

`git fetch` : Download new commits (doesn't modify working directory)

```
sequenceDiagram
    participant Local
    participant Remote

    Local->>Remote: Any new commits?
    Remote-->>Local: Here's C4, C5
    Note over Local: Updates origin/main<br/>Local main unchanged
```

Pull vs Fetch

`git fetch` : Download changes, don't merge

```
git fetch origin
# Downloads new commits to origin/main
# Your local main is unchanged
```

`git pull` : Download + merge (fetch + merge)

```
git pull origin main
# Equivalent to:
git fetch origin
git merge origin/main
```

Visual comparison:

```
graph LR
    subgraph "git fetch"
```

Push: Sending Your Work

After `git push origin main`:

```
graph RL
    subgraph "Remote (GitHub)"
        OR[origin/main] --> RC3[C3]
    end

    subgraph "Local"
        LM[main] --> RC3
        ORM[origin/main] -.->|tracking| RC3
    end

    RC3 --> C2[C2]
    C2 --> C1[C1]

    style OR fill:#ffe1f5
    style LM fill:#e8f5e9
    style ORM fill:#fff4e1
```

Push and Pull

Push changes:

```
# First time (set upstream)
git push -u origin main

# Subsequent pushes
git push
```

Pull changes:

```
# Fetch and merge
git pull

# Equivalent to:
git fetch
git merge origin/main
```

Best practice: Always pull before push

Collaboration Workflow

Standard flow:

```
# 1. Pull latest changes
git pull

# 2. Create feature branch
git checkout -b feature-new-endpoint

# 3. Make changes and commit
git add .
git commit -m "Add new endpoint"

# 4. Push branch
git push -u origin feature-new-endpoint

# 5. Create Pull Request on GitHub

# 6. After review, merge on GitHub

# 7. Update local main
```

Collaboration Workflow Visualized

```
sequenceDiagram
    participant A as Alice (Developer)
    participant LA as Alice's Local
    participant GH as GitHub (Remote)
    participant LB as Bob's Local
    participant B as Bob (Reviewer)

    A->>LA: git checkout -b feature
    A->>LA: Make changes
    A->>LA: git commit
    LA->>GH: git push origin feature

    A->>GH: Create Pull Request
    GH-->>B: Email notification
    B->>GH: Review code
    B->>GH: Request changes

    A->>LA: Fix issues
    LA->>GH: git push (updates PR)
    B->>GH: Approve PR
    GH->>GH: Merge feature → main

    A->>LA: git checkout main
    LA->>GH: git pull
    B->>LB: git pull
```

Complete Git Workflow: Feature Development

```
gitGraph
    commit id: "C1: Initial"
    commit id: "C2: Add models"

    branch feature-login
    checkout feature-login
    commit id: "C3: Add login form"
    commit id: "C4: Add validation"

    checkout main
    commit id: "C5: Update README"

    checkout feature-login
    commit id: "C6: Add tests"

    checkout main
    merge feature-login id: "M1: Merge feature-login"

    commit id: "C7: Deploy to prod"
```

Pull Requests

What is a PR?

- Request to merge your branch
- Code review mechanism
- Discussion platform

Creating a PR (GitHub):

1. Push branch to GitHub
2. Visit repository on GitHub
3. Click "Compare & pull request"
4. Add description
5. Request reviewers
6. Address feedback

Common Git Commands Summary

```
git init                          # Initialize repository
git status                        # Check status
git add <file>                   # Stage files
git commit -m "message"          # Commit changes
git log                          # View history
git diff                         # View changes
git branch                       # List branches
git checkout -b <branch>         # Create and switch branch
git merge <branch>               # Merge branch
git push                         # Push to remote
git pull                         # Pull from remote
git clone <url>                  # Clone repository
```

Git Best Practices

1. Commit often, push regularly
2. Write clear commit messages
3. Use branches for features
4. Pull before push
5. Don't commit sensitive data
6. Use `.gitignore`
7. Review before committing
8. Keep commits focused

Calling External APIs from Python

Why call external APIs?

- Weather data
- Currency exchange rates
- News aggregation
- Translation services
- AI/ML services (LLMs, vision)
- Social media data

Tools:

- `requests` : Synchronous HTTP library
- `httpx` : Modern async/sync library

Using requests Library

Basic GET request:

```
import requests

response = requests.get("https://api.github.com/users/octocat")

print(response.status_code)  # 200
print(response.json())       # Parsed JSON
print(response.headers)      # Response headers
```

POST request:

```
data = {"name": "Alice", "email": "alice@example.com"}

response = requests.post(
    "https://api.example.com/users",
    json=data
)
```

Request Headers and Authentication

Custom headers:

```
headers = {
    "Authorization": "Bearer YOUR_TOKEN",
    "Content-Type": "application/json"
}

response = requests.get(
    "https://api.example.com/data",
    headers=headers
)
```

API Key authentication:

```
params = {"api_key": "YOUR_API_KEY"}

response = requests.get(
    "https://api.openweathermap.org/data/2.5/weather",
```

Error Handling

Always handle errors:

```
import requests

try:
    response = requests.get("https://api.example.com/data", timeout=5)
    response.raise_for_status()  # Raises exception for 4xx/5xx
    data = response.json()
    print(data)
except requests.exceptions.Timeout:
    print("Request timed out")
except requests.exceptions.HTTPError as e:
    print(f"HTTP error: {e}")
except requests.exceptions.RequestException as e:
    print(f"Error: {e}")
```

Retry Logic

Implement retries for reliability:

```
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

session = requests.Session()

retry = Retry(
    total=3,
    backoff_factor=1,
    status_forcelist=[429, 500, 502, 503, 504]
)

adapter = HTTPAdapter(max_retries=retry)
session.mount("https://", adapter)

response = session.get("https://api.example.com/data")
```

Rate Limiting

Respect API rate limits:

```python
import time
import requests

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

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

    return results

urls = [f"https://api.example.com/item/{i}" for i in range(10)]
data = call_api_with_rate_limit(urls, calls_per_second=2)
```

Pagination

Handle paginated responses:

```
def fetch_all_pages(base_url):
    all_data = []
    page = 1

    while True:
        response = requests.get(f"{base_url}?page={page}")
        data = response.json()

        if not data:
            break

        all_data.extend(data)
        page += 1

    return all_data

users = fetch_all_pages("https://api.example.com/users")
```

Using httpx Library

Modern alternative to requests:

```
import httpx

# Synchronous
response = httpx.get("https://api.github.com/users/octocat")
print(response.json())

# Async
import asyncio

async def fetch_data():
    async with httpx.AsyncClient() as client:
        response = await client.get("https://api.github.com/users/octocat")
        return response.json()

data = asyncio.run(fetch_data())
```

Async API Calls

Fetch multiple URLs concurrently:

```
import asyncio
import httpx

async def fetch_all(urls):
    async with httpx.AsyncClient() as client:
        tasks = [client.get(url) for url in urls]
        responses = await asyncio.gather(*tasks)
        return [r.json() for r in responses]

urls = [
    "https://api.github.com/users/octocat",
    "https://api.github.com/users/torvalds",
]

data = asyncio.run(fetch_all(urls))
```

Much faster than sequential requests!

Integrating APIs with FastAPI

Example: Weather API endpoint

```
from fastapi import FastAPI, HTTPException
import requests

app = FastAPI()

@app.get("/weather/{city}")
def get_weather(city: str):
    api_key = "YOUR_API_KEY"
    url = f"https://api.openweathermap.org/data/2.5/weather"

    try:
        response = requests.get(url, params={
            "q": city,
            "appid": api_key,
            "units": "metric"
        })
        response.raise_for_status()
        return response.json()
```

Integrating LLM APIs - Gemini

Setup:

```
import os
from google import genai
from fastapi import FastAPI

app = FastAPI()

client = genai.Client(api_key=os.environ['GEMINI_API_KEY'])

@app.post("/generate")
def generate_text(prompt: str):
    response = client.models.generate_content(
        model="models/gemini-2.0-flash-exp",
        contents=prompt
    )
    return {"response": response.text}
```

Test:

LLM API - Text Understanding

Sentiment analysis endpoint:

```
from pydantic import BaseModel

class TextInput(BaseModel):
    text: str

@app.post("/sentiment")
def analyze_sentiment(input: TextInput):
    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
```

LLM API - Structured Output

Extract entities as JSON:

```
import json

@app.post("/extract-entities")
def extract_entities(input: TextInput):
    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
    )

    try:
        entities = json.loads(response.text)
        return entities
```

LLM API - Image Analysis

Analyze uploaded images:

```
from fastapi import UploadFile, File
from PIL import Image
import io

@app.post("/analyze-image")
async def analyze_image(
    file: UploadFile = File(...),
    prompt: str = "Describe this image"
):
    contents = await file.read()
    image = Image.open(io.BytesIO(contents))

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

    return {
        "filename": file.filename,
```

Streaming Responses

Stream LLM output in real-time:

```
from fastapi.responses import StreamingResponse

@app.post("/stream")
async def stream_response(input: TextInput):
    def generate():
        response = client.models.generate_content(
            model="models/gemini-2.0-flash-exp",
            contents=input.text,
            config={"stream": True}
        )

        for chunk in response:
            yield chunk.text

    return StreamingResponse(generate(), media_type="text/plain")
```

Client sees text appear progressively

Cost Optimization

Strategies:

1. **Cache responses:**

```python
from functools import lru_cache

@lru_cache(maxsize=100)
def get_llm_response(prompt: str):
    response = client.models.generate_content(
        model="models/gemini-2.0-flash-exp",
        contents=prompt
    )
    return response.text
```

2. **Use cheaper models when possible**
3. **Limit output length**
4. **Batch similar requests**

Building a Translation API

Complete example:

```
from fastapi import FastAPI
from pydantic import BaseModel
from google import genai
import os

app = FastAPI()
client = genai.Client(api_key=os.environ['GEMINI_API_KEY'])

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

@app.post("/translate")
def translate(req: TranslationRequest):
    prompt = f"Translate to {req.target_language}: {req.text}"

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

    return {
        "original": req.text,
```

Building a Summarization API

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

@app.post("/summarize")
def summarize(req: SummarizeRequest):
    prompt = f"""
    Summarize in {req.max_sentences} sentences:

    {req.text}
    """

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

    return {
        "original_length": len(req.text),
        "summary": response.text
    }
```

Building a QA API

Question answering with context:

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

@app.post("/qa")
def answer_question(req: QARequest):
    prompt = f"""
    Context: {req.context}

    Question: {req.question}

    Answer:
    """

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

    return {
        "question": req.question,
```

Environment Variables

Never hardcode API keys!

Create `.env` file:

```
GEMINI_API_KEY=your_key_here
OPENWEATHER_API_KEY=your_key_here
```

Load in Python:

```
from dotenv import load_dotenv
import os

load_dotenv()

gemini_key = os.getenv("GEMINI_API_KEY")
weather_key = os.getenv("OPENWEATHER_API_KEY")
```

Add `.env` to `.gitignore`!

Error Handling Best Practices

```python
from fastapi import HTTPException

@app.post("/generate")
def generate(prompt: str):
    if not prompt.strip():
        raise HTTPException(status_code=400, detail="Prompt cannot be empty")

    try:
        response = client.models.generate_content(
            model="models/gemini-2.0-flash-exp",
            contents=prompt
        )
        return {"response": response.text}
    except Exception as e:
        raise HTTPException(
            status_code=500,
            detail=f"LLM API error: {str(e)}"
        )
```

Logging and Monitoring

Track API usage:

```
import logging

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

@app.post("/generate")
def generate(prompt: str):
    logger.info(f"Received prompt: {prompt[:50]}...")

    try:
        response = client.models.generate_content(
            model="models/gemini-2.0-flash-exp",
            contents=prompt
        )
        logger.info(f"Generated response length: {len(response.text)}")
        return {"response": response.text}
    except Exception as e:
```

Testing API Integrations

Mock external APIs:

```
from unittest.mock import Mock, patch

def test_weather_endpoint():
    mock_response = Mock()
    mock_response.json.return_value = {"temp": 20}

    with patch('requests.get', return_value=mock_response):
        response = client.get("/weather/London")
        assert response.status_code == 200
        assert response.json()["temp"] == 20
```

Complete Example: Text Tools API

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

app = FastAPI(title="Text Tools API")
client = genai.Client(api_key=os.environ['GEMINI_API_KEY'])

class TextRequest(BaseModel):
    text: str

@app.post("/sentiment")
def sentiment(req: TextRequest):
    prompt = f"Sentiment (Positive/Negative/Neutral): {req.text}"
    response = client.models.generate_content(
        model="models/gemini-2.0-flash-exp",
        contents=prompt
    )
    return {"sentiment": response.text}

@app.post("/summarize")
def summarize(req: TextRequest):
    prompt = f"Summarize in 2 sentences: {req.text}"
    response = client.models.generate_content(
        model="models/gemini-2.0-flash-exp",
        contents=prompt
    )
    return {"summary": response.text}
```

What We've Learned

Git Fundamentals:

- Version control concepts
- Basic workflow (add, commit, push, pull)
- Branching and merging
- Collaboration with GitHub

External API Integration:

- Using requests and httpx
- Error handling and retries
- Rate limiting and pagination

LLM API Integration:

Best Practices Summary

Git:

- Commit often, meaningful messages
- Use branches for features
- Pull before push
- Never commit secrets

API Integration:

- Handle errors gracefully
- Implement retries
- Respect rate limits
- Cache when possible
- Use environment variables

Questions?

Next week: Active Learning

Lab: Build complete Text Tools API