GitHub Automation & GraphQL

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

Why Automate GitHub?

Git is for version control. **GitHub** is a collaboration platform.

We spend 50% of our time coding, and 50% managing the process:

- Creating repos, setting up protection rules.
- Triaging issues (Bug? Feature? Duplicate?).
- Reviewing PRs (Style checks, logic errors).
- Managing releases.

Automation turns "Process" into "Code".

- *Infrastructure as Code* (Terraform) → *Repo as Code* (PyGithub).
- *Manual Review → AI Reviewer*.

The GitHub API Ecosystem

1. **REST API (v3)**:
    - Standard resource-based access.
    - `GET /repos/{owner}/{repo}/issues`
    - Good for simple tasks, scripts.
    - Problem: Over-fetching (getting too much data).
2. **GraphQL API (v4)**:
    - Query language for your API.
    - Ask for exactly what you need.
    - Single endpoint: `https://api.github.com/graphql`
3. **Webhooks**:
    - "Don't call us, we'll call you."
    - GitHub notifies your server when events happen (Push, PR opened).

Authentication

1. **Personal Access Tokens (PATs)**:
   - **Classic**: Broad scopes (`repo`, `user`). Risky if leaked.
   - **Fine-grained**: Scoped to specific repos and permissions (Read-only on Issues). **Use these!**
2. **OAuth Apps**:
   - "Login with GitHub". User grants your app permission.
3. **GitHub Apps**:
   - Installed on organizations/repos. First-class citizens.
   - Have their own identity (bot).

Python Tooling: PyGithub

Object-Oriented wrapper for the REST API.

```
pip install PyGithub
```

Listing My Repos:

```python
from github import Github
import os

# Auth
g = Github(os.getenv("GITHUB_TOKEN"))

# Get User
user = g.get_user()

# Iterate
for repo in user.get_repos():
    print(repo.name, repo.stargazers_count)
```

Use Case 1: The "Repo Scaffolder"

Problem: Every new project needs:

- A private repo.
- A `README.md`.
- A `.gitignore`.
- Branch protection (no force push to main).

Solution:

```
repo = user.create_repo("new-project", private=True)

repo.create_file("README.md", "init", "# New Project")
repo.create_file(".gitignore", "init", "__pycache__/")

# Branch Protection
branch = repo.get_branch("main")
branch.edit_protection(strict=True, enforce_admins=True)
```

Introduction to GraphQL

REST Problem:

To get "The latest 3 issues of my top 5 repos":

1. `GET /user/repos` (Returns 100s of fields per repo).
2. For each repo: `GET /repos/ ... /issues` (N+1 problem).

GraphQL Solution: Single Request.

```
query {
  viewer {
    repositories(first: 5, orderBy: {field: STARGAZERS, direction: DESC}) {
      nodes {
        name
        issues(last: 3) {
          nodes {
            title
            createdAt
          }
        }
      }
```

Using GraphQL in Python

Simple HTTP POST request.

```
import requests

query = """
query {
  viewer {
    login
  }
}
"""

headers = {"Authorization": f"Bearer {token}"}
response = requests.post(
    "https://api.github.com/graphql",
    json={'query': query},
    headers=headers
)

print(response.json())
```

Use Case 2: AI Issue Triaging

Scenario: User opens issue "App crashes on start".
Goal: Auto-label as `bug` and assign to `developer-x`.

Workflow:

1. Fetch new issues without labels.
2. Send `title` + `body` to **Gemini API**.
3. Prompt: "Classify this as [bug, feature, question]".
4. Call GitHub API to add label.

```
# Pseudo-code
for issue in repo.get_issues(state='open'):
    if not issue.labels:
        category = gemini.classify(issue.title, issue.body)
        issue.add_to_labels(category)
        print(f"Labeled {issue.number} as {category}")
```

Use Case 3: The AI Code Reviewer

The Holy Grail of AI in DevTools.

Process:

1. Get Pull Request object.
2. Get the **Diff** (changes).
    - `pr.get_files()`
3. For each file:
    - Get `patch` (the +/- lines).
    - Send to LLM: "Review this code patch for bugs."
4. Post comment on specific line.

Fetching the Diff

```
pr = repo.get_pull(123)

for file in pr.get_files():
    print(f"File: {file.filename}")
    print(f"Status: {file.status}") # added/modified
    print(f"Patch:\n{file.patch}")

    # Send 'file.patch' to LLM
```

Posting a Review Comment:

```
pr.create_review_comment(
    body="⚠️ Potential SQL Injection here. Use parameterized queries.",
    commit_id=list(pr.get_commits())[-1],
    path=file.filename,
    position=5 # Line number in diff
)
```

Webhooks: Reacting to Events

Scripts only run when you execute them. **Webhooks** run when things happen.

Setup:

1. Go to Repo Settings → Webhooks.
2. Payload URL: `https://my-api.onrender.com/webhook`.
3. Content type: `application/json`.
4. Select events: `Pull requests`, `Issues`.

Handling in FastAPI:

```
@app.post("/webhook")
async def handle_github_event(request: Request):
    payload = await request.json()

    if payload.get('action') == 'opened' and 'issue' in payload:
        issue_body = payload['issue']['body']
        # Trigger AI Triage Bot
```

Rate Limiting & Best Practices

1. **Rate Limits**:
    - REST: 5,000 requests/hour per token.
    - GraphQL: 5,000 points/hour.
    - *Always* handle 429/403 errors.
2. **Pagination**:
    - Don't assume `get_repos()` returns everything.
    - PyGithub handles this automatically (mostly).
    - In GraphQL, use `cursors`.
3. **Secrets**:
    - Never commit `.env`.
    - Use `python-dotenv`.

Resources

- **GitHub GraphQL Explorer**: https://docs.github.com/en/graphql/overview/explorer
- **PyGithub Docs**: https://pygithub.readthedocs.io/
- **GitHub REST API**: https://docs.github.com/en/rest

Questions?

Lab: Build the "AI PR Reviewer" Bot!