МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ ЭЛЕКТРОННО-ИНФОРМАЦИОННЫХ СИСТЕМ

Кафедра интеллектуальных информационных технологий

Отчет по лабораторной работе No4

Специальность ПО11(о)

Выполнил
И. А. Головач,
студент группы ПО11

Проверил
А. А. Крощенко,
ст. преп. кафедры ИИТ,
«26» апрель 2025 г.

Брест 2025

Вариант 5

Цель работы: научиться работать с Github API, приобрести практические навыки написания программ для работы с REST API или GraphQL API

Общее задание: используя Github API, реализовать предложенное задание на языке Python. Выполнить визуализацию результатов, с использованием графика или отчета. Можно использовать как REST API (рекомендуется), так и GraphQL

Задание:
Построение карты взаимодействий разработчика на GitHub
Условие:
Напишите Python-скрипт, который:
1. Запрашивает у пользователя имя пользователя GitHub.
2. Использует API GitHub для получения списка всех репозиториев, в которых этот пользователь:
- Делал коммиты
- Открывал pull requests
- Создавал issues
- Ставил звёзды

3. Для каждого репозитория находит других разработчиков, с которыми пользователь взаимодействовал (например, авторы коммитов, ревьюеры pull requests, авторы issues).
4. Создаёт граф связей между пользователем и другими разработчиками.
5. Визуализирует граф с помощью networkx и matplotlib:
- Узлы (nodes) — пользователи
- Рёбра (edges) — взаимодействия (коммиты, PR, issues, звёзды)

6. Сохраняет граф в формате PNG/JPEG.

Выполнение:
Код программы:

main.py:
```
from api import fetch_github_data, generate_report
from graph_builder import build_interaction_graph
from visualizer import visualize_and_save_graph
from json_saver import save_graph_to_json

def get_github_username() -> str:
    return input("Введите имя пользователя GitHub: ").strip()

def main():
    username = get_github_username()
    print(f"Анализируем взаимодействия пользователя {username}...")

    # Получение данных через API
```

```
    repos_data = fetch_github_data(username)
    if not repos_data:
        print("Не удалось получить данные от GitHub API.")
        return

    # Генерация и вывод отчёта
    report = generate_report(username, repos_data, f"{username}_github_report.txt")
    print("\n" + report)
    print(f"\nТекстовый отчёт сохранён в {username}_github_report.txt")

    # Построение графа взаимодействий
    graph = build_interaction_graph(username, repos_data)
    print(f"\nНайдено {len(graph.nodes)-1} связанных разработчика.")

    # Сохранение графа в JSON
    save_graph_to_json(graph, f"{username}_github_network.json")

    # Визуализация графа
    visualize_and_save_graph(graph, f"{username}_github_network.png")

if __name__ == "__main__":
    main()
```

api.py:

```
import requests
from const import GITHUB_TOKEN, GITHUB_API_URL


def fetch_github_data(username: str):
    """Fetch GitHub user interaction data including repos, contributions and stars."""
    headers = {"Authorization": f"token {GITHUB_TOKEN}"} if GITHUB_TOKEN else {}

    try:
        user_repos = _fetch_user_repositories(username, headers)
        if not user_repos:
            print("Пользователь не имеет репозиториев.")
            return None

        contributed_prs = _fetch_user_pull_requests(username, headers)
        data = _initialize_data_structure()

        _process_user_repositories(user_repos, username, headers, data)
        _process_contributions(contributed_prs, headers, data)
        _fetch_user_stars(username, headers, data)

        return data

    except requests.exceptions.RequestException as e:
        print(f"Ошибка при запросе к GitHub API: {str(e)}")
        return None


def _fetch_user_repositories(username: str, headers: dict) -> list:
    """Fetch all user repositories including forks."""
    url = f"{GITHUB_API_URL}/users/{username}/repos?type=all"
    response = requests.get(url, headers=headers, timeout=10)
```

```
    if response.status_code != 200:
        print(f"Ошибка при запросе репозиториев: {response.status_code}")
        print(f"Ответ сервера: {response.text}")
        return []

    return response.json()


def _fetch_user_pull_requests(username: str, headers: dict) -> list:
    """Fetch user's pull requests in other repositories."""
    url = f"{GITHUB_API_URL}/search/issues?q=author:{username}+type:pr"
    response = requests.get(url, headers=headers, timeout=10)
    return response.json().get("items", []) if response.status_code == 200 else []


def _initialize_data_structure() -> dict:
    """Initialize the data structure for storing GitHub interactions."""
    return {
        "repos": [],  # Собственные репозитории
        "contributions": [],  # Участие в чужих репозиториях
        "stars": []  # Звёзды
    }


def _process_user_repositories(repos: list, username: str,
                               headers: dict, data: dict) -> None:
    """Process user repositories and collect interaction data."""
    for repo in repos:
        repo_name = repo["full_name"]
        repo_data = {
            "name": repo_name,
            "commits": _fetch_repository_commits(repo_name, username, headers),
            "pull_requests": _fetch_repository_pull_requests(repo_name, headers),
            "issues": _fetch_repository_issues(repo_name, headers)
        }
        data["repos"].append(repo_data)

        if repo.get("fork"):
            _process_forked_repository(repo, repo_name, headers, data)


def _process_forked_repository(repo: dict, repo_name: str,
                               headers: dict, data: dict) -> None:
    """Process forked repository and find pull requests to parent."""
    parent = repo.get("parent")
    if parent:
        parent_repo = parent["full_name"]
        prs = _fetch_pull_requests_from_fork(repo_name, parent_repo, headers)
        if prs:
            data["contributions"].extend(prs)


def _process_contributions(prs: list, headers: dict, data: dict) -> None:
    """Process user contributions to other repositories."""
    for pr in prs:
        repo_url = pr["repository_url"]
        repo_name = "/".join(repo_url.split("/")[-2:])
        pr_data = {
```

```
            "repo": repo_name,
            "user": pr["user"]["login"],
            "reviewers": _get_pull_request_reviewers(repo_name, pr["number"], headers)
        }
        data["contributions"].append(pr_data)


def _fetch_user_stars(username: str, headers: dict, data: dict) -> None:
    """Fetch repositories starred by the user."""
    url = f"{GITHUB_API_URL}/users/{username}/starred"
    response = requests.get(url, headers=headers, timeout=10)
    if response.status_code == 200:
        data["stars"] = [repo["owner"]["login"] for repo in response.json()]


def _fetch_repository_commits(repo_name: str, username: str,
                              headers: dict) -> list:
    """Fetch commits for a specific repository."""
    url = f"{GITHUB_API_URL}/repos/{repo_name}/commits?author={username}"
    response = requests.get(url, headers=headers, timeout=10)
    if response.status_code == 200:
        return list({commit["author"]["login"]
                     for commit in response.json()
                     if commit.get("author")})
    return []


def _fetch_repository_pull_requests(repo_name: str, headers: dict) -> list:
    """Fetch pull requests for a specific repository."""
    url = f"{GITHUB_API_URL}/repos/{repo_name}/pulls?state=all"
    response = requests.get(url, headers=headers, timeout=10)
    if response.status_code == 200:
        return [{
            "user": pr["user"]["login"],
            "reviewers": [r["login"] for r in pr.get("requested_reviewers", [])]
        } for pr in response.json()]
    return []


def _fetch_pull_requests_from_fork(fork_name: str,
                                   parent_repo: str,
                                   headers: dict) -> list:
    """Fetch pull requests from a fork to its parent repository."""
    owner, branch = fork_name.split('/')
    url = (f"{GITHUB_API_URL}/repos/{parent_repo}/pulls"
           f"?state=all&head={owner}:{branch}")
    response = requests.get(url, headers=headers, timeout=10)
    if response.status_code == 200:
        return [{
            "repo": parent_repo,
            "user": pr["user"]["login"],
            "reviewers": _get_pull_request_reviewers(parent_repo, pr["number"],
headers)
        } for pr in response.json()]
    return []


def _get_pull_request_reviewers(repo_name: str,
                                pr_number: int,
```

```
                                headers: dict) -> list:
    """Fetch reviewers for a specific pull request."""
    url = f"{GITHUB_API_URL}/repos/{repo_name}/pulls/{pr_number}/reviews"
    response = requests.get(url, headers=headers, timeout=10)
    if response.status_code == 200:
        return list({review["user"]["login"]
                     for review in response.json()
                     if review.get("user")})
    return []


def _fetch_repository_issues(repo_name: str, headers: dict) -> list:
    """Fetch issues for a specific repository."""
    url = f"{GITHUB_API_URL}/repos/{repo_name}/issues?state=all"
    response = requests.get(url, headers=headers, timeout=10)
    if response.status_code == 200:
        return list({issue["user"]["login"]
                     for issue in response.json()
                     if issue.get("user")})
    return []


def generate_report(username: str, data: dict, filename: str) -> str:
    """Generate and save a text report of user interactions."""
    report = f"Отчёт по взаимодействиям пользователя {username}:\n\n"

    # Коммиты
    commit_repos = {repo["name"] for repo in data["repos"] if repo["commits"]}
    report += f"Репозитории с коммитами ({len(commit_repos)}):\n"
    report += "\n".join(f"- {repo}" for repo in commit_repos) + "\n\n"

    # Pull Requests
    pr_repos = {repo["name"] for repo in data["repos"] if repo["pull_requests"]}
    pr_repos.update(contrib["repo"] for contrib in data["contributions"])
    report += f"Репозитории с Pull Requests ({len(pr_repos)}):\n"
    report += "\n".join(f"- {repo}" for repo in pr_repos) + "\n\n"

    # Issues
    issue_repos = {repo["name"] for repo in data["repos"] if repo["issues"]}
    report += f"Репозитории с Issues ({len(issue_repos)}):\n"
    report += "\n".join(f"- {repo}" for repo in issue_repos) + "\n\n"

    # Stars
    report += f"Репозитории с звёздами ({len(data['stars'])}):\n"
    report += "\n".join(f"- {owner}" for owner in data["stars"]) + "\n\n"

    with open(filename, "w", encoding="utf-8") as f:
        f.write(report)

    return report
```

const.py:

```
import os

GITHUB_TOKEN = os.getenv("GITHUB_TOKEN")
GITHUB_API_URL = "https://api.github.com"
```

graph_builder.py:

```
import networkx as nx


def build_interaction_graph(username: str, data: dict) -> nx.Graph:
    """
    Builds an interaction graph for a GitHub user based on their activities.

    Args:
        username: GitHub username
        data: Dictionary containing user's repositories, contributions and stars

    Returns:
        NetworkX Graph object representing user's interactions
    """
    G = nx.Graph()
    G.add_node(username, type="user")

    _add_repository_interactions(G, username, data["repos"])
    _add_contribution_interactions(G, username, data["contributions"])
    _add_star_interactions(G, username, data["stars"])

    return G


def _add_repository_interactions(graph: nx.Graph, username: str, repos: list) -> None:
    """Add interactions from user's own repositories."""
    for repo in repos:
        _add_commit_interactions(graph, username, repo["commits"])
        _add_pull_request_interactions(graph, username, repo["pull_requests"])
        _add_issue_interactions(graph, username, repo["issues"])


def _add_contribution_interactions(graph: nx.Graph, username: str, contributions: list)
-> None:
    """Add interactions from contributions to other repositories."""
    for contribution in contributions:
        _add_contribution_edge(graph, username, contribution["user"], "contribution")
        _add_reviewer_edges(graph, username, contribution["reviewers"],
"contribution_review")


def _add_star_interactions(graph: nx.Graph, username: str, stars: list) -> None:
    """Add interactions from starred repositories."""
    for owner in stars:
        _add_star_edge(graph, username, owner)


def _add_commit_interactions(graph: nx.Graph, username: str, commits: list) -> None:
    """Add commit interactions to the graph."""
    for author in commits:
        _add_edge_if_valid(graph, username, author, "commit")


def _add_pull_request_interactions(graph: nx.Graph, username: str, pull_requests: list)
-> None:
    """Add pull request interactions to the graph."""
    for pr in pull_requests:
```

```
        _add_edge_if_valid(graph, username, pr["user"], "pull_request")
        _add_reviewer_edges(graph, username, pr["reviewers"], "pr_review")


def _add_issue_interactions(graph: nx.Graph, username: str, issues: list) -> None:
    """Add issue interactions to the graph."""
    for user in issues:
        _add_edge_if_valid(graph, username, user, "issue")


def _add_reviewer_edges(graph: nx.Graph, username: str, reviewers: list, edge_type:
str) -> None:
    """Add reviewer edges to the graph."""
    for reviewer in reviewers:
        _add_edge_if_valid(graph, username, reviewer, edge_type)


def _add_contribution_edge(graph: nx.Graph, username: str, user: str, edge_type: str) -
> None:
    """Add a contribution edge to the graph."""
    _add_edge_if_valid(graph, username, user, edge_type)


def _add_star_edge(graph: nx.Graph, username: str, owner: str) -> None:
    """Add a star edge to the graph."""
    _add_edge_if_valid(graph, username, owner, "star")


def _add_edge_if_valid(graph: nx.Graph, source: str, target: str, edge_type: str) ->
None:
    """
    Add an edge to the graph if target is valid and not the same as source.

    Args:
        graph: NetworkX Graph to add edge to
        source: Source node
        target: Target node
        edge_type: Type of interaction/edge
    """
    if target and target != source:
        graph.add_edge(source, target, type=edge_type)
```

json_saver.py:

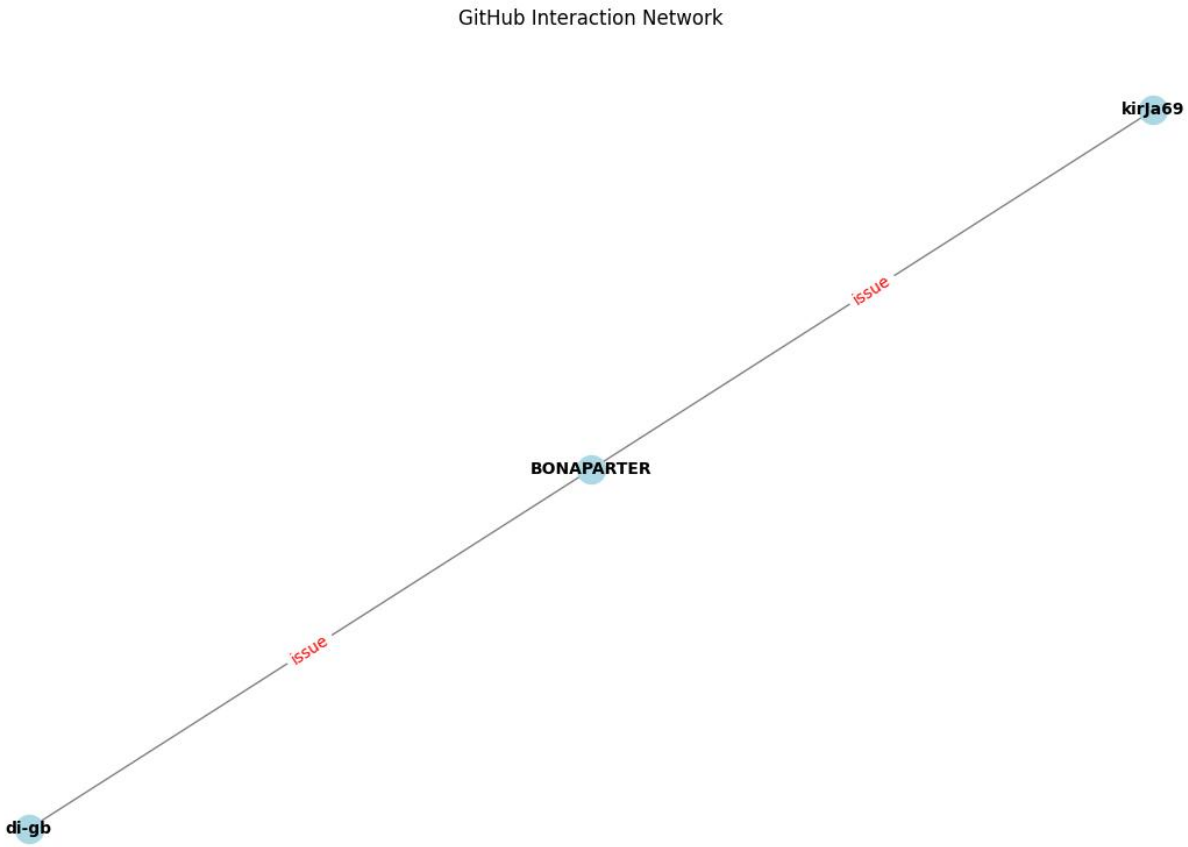
```
import json

def save_graph_to_json(graph, filename: str):
    graph_data = {
        "nodes": [{"id": node, "type": graph.nodes[node].get("type", "unknown")} for
node in graph.nodes],
        "edges": [{"source": u, "target": v, "type": d["type"]} for u, v, d in
graph.edges(data=True)],
    }
    with open(filename, "w", encoding="utf-8") as file:
        json.dump(graph_data, file, ensure_ascii=False, indent=4)
    print(f"Граф сохранён в {filename}")
```

visualizer.py:

```
import matplotlib.pyplot as plt
import networkx as nx

def visualize_and_save_graph(G, filename: str):
    plt.figure(figsize=(12, 8))
    pos = nx.spring_layout(G)

    # Рисуем узлы
    nx.draw_networkx_nodes(G, pos, node_size=300, node_color="lightblue")
    nx.draw_networkx_labels(G, pos, font_size=10, font_weight="bold")

    # Рисуем рёбра
    edge_labels = nx.get_edge_attributes(G, 'type')
    nx.draw_networkx_edges(G, pos, width=1.0, alpha=0.5)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_color='red')

    plt.title("GitHub Interaction Network")
    plt.axis("off")
    plt.tight_layout()

    # Сохранение графика
    plt.savefig(filename)
    print(f"Визуализация графа сохранена в {filename}")
    plt.show()
```

Результаты работы программы:

```
Введите имя пользователя GitHub: BONAPARTER
Анализируем взаимодействия пользователя BONAPARTER...

Отчёт по взаимодействиям пользователя BONAPARTER:

Репозитории с коммитами (8):
- BONAPARTER/spp_po11
- BONAPARTER/Mobile-development
- BONAPARTER/task_manager
- di-gb/tg_bot_test
- BONAPARTER/Tests
- BONAPARTER/Json_data
- BONAPARTER/KPO
- BONAPARTER/Tic-tac-toe

Репозитории с Pull Requests (2):
- kroschenko/spp_po11
- di-gb/tg_bot_test

Репозитории с Issues (1):
- di-gb/tg_bot_test

Репозитории с звёздами (0):



Текстовый отчёт сохранён в BONAPARTER_github_report.txt

Найдено 2 связанных разработчика.
Граф сохранён в BONAPARTER_github_network.json
Визуализация графа сохранена в BONAPARTER_github_network.png
```



BONAPARTER_github_network.json BONAPARTER_github_report.txt

```
Отчёт по взаимодействиям пользователя BONAPARTER:

Репозитории с коммитами (8):
- BONAPARTER/spp_po11
- BONAPARTER/Mobile-development
- BONAPARTER/task_manager
- di-gb/tg_bot_test
- BONAPARTER/Tests
- BONAPARTER/Json_data
- BONAPARTER/KPO
- BONAPARTER/Tic-tac-toe

Репозитории с Pull Requests (2):
- kroschenko/spp_po11
- di-gb/tg_bot_test

Репозитории с Issues (1):
- di-gb/tg_bot_test

Репозитории с звёздами (0):
```

Вывод: научился работать с Github API, а также визуализировать зависимости между пользователями.