Факультет бизнеса
Кафедра экономической информатики

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ РАСЧЕТНО-ГРАФИЧЕСКОГО ЗАДАНИЯ

Дисциплина: «Разработка программных приложений»

Тема: «Разработка Flask приложения и workflow на GitHub Actions»

Направление: 38.03.05 «Бизнес-информатика»
Год набора: 2024

Новосибирск, 2024

Содержание

Выполнение

Задание выполняется в парах. Инструкция по установке Python библиотек расположена в главе: «Инструкция по установке Python библиотек»

Варианты

Вариант 1

Задание

Разработать Flask-приложение (API) для управления телефонными контактами, а также настроить автоматический workflow в GitHub Actions для генерации и публикации документации на GitHub Pages.

Требования к Flask-приложению:

1. Эндпоинты для работы с телефонными контактами:
    1) POST /contacts: создание нового контакта;
    2) GET /contacts/<id>: получение информации о контакте по его идентификатору;
    3) DELETE /contacts/<id>: удаление контакта.
2. Хранение данных: для хранения данных о контактах используйте **Python-словарь**.
3. Документация API: документация API должна быть сгенерирована в формате OpenAPI с использованием библиотеки Flasgger.

Требования к workflow в GitHub Actions

1. Workflow должен запускаться автоматически при push в main.
2. Проверка безопасности кода с помощью Bandit.
3. Генерация OpenAPI документации:
    1) workflow должен автоматически извлекать описание эндпоинтов из кода Flask-приложения и создавать файл спецификации OpenAPI (например, openapi.yaml);
    2) установка redocly;
    3) генерация страницы ReDoc с помощью redocly.
4. Публикация сгенерированной страницы документации с помощью redocly на GitHub Pages.

Инструменты для реализации задания

ReDoc — это инструмент для генерации интерактивной веб-документации из файла описания API в формате OpenAPI. Для установки redocly добавьте в пайплайн следующие шаги:

```
- name: Install Node
  uses: actions/setup-node@v4
  with:
    node-version: 18

- name: Install Redocly
  run: |
    npm i -g @redocly/cli@latest
```

Для генерации документации выполните команду:

```
redocly build-docs {название файла}
```

Flasgger — это библиотека для описания API прямо в коде Python и автоматической генерации JSON спецификации API.

Чтобы преобразовать JSON спецификацию Flasgger в формат OpenAPI и сохранить её в openapi.yaml, используйте следующий скрипт:

```
import yaml
from app import app  # app — это Flask объект в вашем Flask-приложении

def generate_openapi_yaml():
    with app.test_client() as client:
        response = client.get('/apispec_1.json')

        if response.status_code == 200 and response.is_json:
            openapi_spec = response.json

            with open("docs/openapi.yaml", "w") as file:
                yaml.dump(openapi_spec, file, default_flow_style=False)
            print("Спецификация OpenAPI сохранена в docs/openapi.yaml")
        else:
            print("Ошибка: не удалось получить спецификацию OpenAPI.")

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

Этот скрипт извлекает JSON-спецификацию и сохраняет её в формате OpenAPI в файл openapi.yaml в директории docs.

Вариант 2

Задание

1. Разработать RESTful API на Flask для анализа текстов, выполняющее операции:
    i. Подсчёт общего количества слов в переданном тексте.
    ii. Определение самых частотных слов в переданном тексте.
2. Реализовать unit-тесты для проверки функциональности.
3. Настроить балансировку нагрузки с помощью Nginx:
    i. Запустить несколько инстансов Flask-приложения на разных портах (например, 5001, 5002, 5003).
    ii. Настроить Nginx для распределения запросов между инстансами.
4. Настроить workflow для CI/CD в GitHub Actions.

Требования RESTful API:

1. Реализовать POST /analyze: принимает текст в JSON формате ({"text": "ваш текст"}) и возвращает количество слов и топ наиболее частотных слов. Настроить workflow для CI/CD в GitHub Actions.

Требования к workflow в GitHub Actions

1) Автоматический запуск unit-тестов при каждом push в ветку main.
2) Проверку безопасности Python-кода с помощью bandit.

Вариант 3

Задание

5. Необходимо разработать RESTful API на Flask для ведения дневника расходов, которое позволяет пользователю добавлять, редактировать, удалять и просматривать свои записи о расходах. Доступ к данным должен быть защищен через авторизацию с использованием **Flask-Login**, а все действия пользователей должны регистрироваться в системе аудита. Настроить workflow для CI/CD в GitHub Actions.

Требования к Flask приложению:

1. Данные о пользователях, записях о расходах и действиях аудита должны храниться в базе данных PostgreSQL.
2. Эндпоинты, которые доступны после авторизации:

1) **Создание записи (POST /add)**: пользователь может добавить новую запись с указанием суммы, категории и описания.

2) **Просмотр записей (GET /list)**: пользователь может просмотреть все свои записи о расходах.

3) **Редактирование записи (POST /edit)**: пользователь может обновить сумму, категорию или описание своих записей.

4) **Удаление записи (POST /delete)**: пользователь может удалить любую из своих записей.

Требования к системе аудита:

1) Систему должна записывает все действия пользователя (например, добавление, изменение и удаление записей).

2) Аудит должен сохранять следующую информацию: идентификатор пользователя, тип действия (добавление, изменение, удаление), время действия и идентификатор измененной записи.

3) Логи аудита должны храниться в базе данных PostgreSQL.

Требования к workflow в GitHub Actions

3) Автоматический запуск unit-тестов при каждом push в ветку main.

4) Проверку безопасности Python-кода с помощью bandit.

Вариант 4

Задание

Необходимо разработать RESTful API на Flask для управления финансовыми подписками пользователя. Приложение должно позволять пользователям добавлять, редактировать, удалять и просматривать свои подписки. Дополнительно нужно разработать bash-хелпер, который автоматизирует настройку и запуск приложения. Настроить workflow для CI/CD в GitHub Actions.

Требования к Flask приложению:

1. Энодпоинты:

1) Создание подписки: пользователь добавляет новую подписку, указывая название, сумму, периодичность списания (например, ежемесячно, ежегодно) и дату начала.

2) Просмотр подписок: пользователь может видеть все свои активные подписки с деталями.

3) Редактирование подписки: пользователь может обновить информацию о сумме, периодичности или дате следующего списания.

4) Удаление подписки: пользователь может удалить подписку.

2. Хранение данных о пользователях, подписках и действиях аудита должно осуществляться в базе данных PostgreSQL.

Требования к bash-хелперу

1. setup_database: настройка базы данных PostgreSQL, создание таблиц для пользователей, подписок и аудита.
2. install_dependencies: создание виртуального окружения Python и установка всех зависимостей приложения.
3. start_app: запуск Flask-приложения.
4. stop_app: остановка Flask-приложения.
5. run_tests: запуск тестов для проверки функциональности приложения.

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.
2. Проверку безопасности Python-кода с помощью bandit.

Вариант 5

Задание

Необходимо разработать RESTful API на Flask для управления финансовыми подписками пользователя.

Реализовать мигратор на Python для Flask-приложения, который запускается автоматически при старте приложения. Мигратор должен использовать changelog-файл для отслеживания миграций.

Настроить workflow для CI/CD в GitHub Actions.

Требования к Flask приложению:

1. Энодпоинты:

1) Создание подписки: пользователь добавляет новую подписку, указывая название, сумму, периодичность списания (например, ежемесячно, ежегодно) и дату начала.

2) Просмотр подписок: пользователь может видеть все свои активные подписки с деталями.

3) Редактирование подписки: пользователь может обновить информацию о сумме, периодичности или дате следующего списания.

4) Удаление подписки: пользователь может удалить подписку.

2. Хранение данных о пользователях, подписках и действиях аудита должно осуществляться в базе данных PostgreSQL.

3. Перед запуском приложения сначала должен отработать мигратор.

Требования к мигратору:

1. **Формат changelog**:

1) Файл changelog в формате yaml, который будет содержать информацию о каждом обновлении схемы базы данных. Пример:

```
- id: 1
  file_path: "migrations/001_create_subscriptions_table.sql"
- id: 2
  file_path: "migrations/002_add_periodicity_column.sql"
```

2) В changelog должны храниться данные о каждом выполненном обновлении: версия, идентификатор миграции, путь до миграции.

2. **Логика работы мигратора**:

1) При запуске Flask-приложения мигратор должен автоматически проверять текущие выполненные миграции в таблице migrations_log и сравнивать их с записями в changelog.

2) Если миграция не была ранее выполнена, то нужно применить ее и добавляйте запись о ней в базу данных migrations_log.

3) Если миграция была выполнена ранее и её состояние в базе данных соответствует changelog, мигратор должен пропускать эту миграцию.

4) Если находятся несоответствия между применёнными миграциями и записями в changelog (например, миграция была выполнена, но затем changelog был изменён), мигратор должен записать в лог сообщение об ошибке, остановить запуск приложения и указать, что база данных находится в несогласованном состоянии.

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.
2. Проверку безопасности Python-кода с помощью bandit.

Вариант 6:

Задание

Необходимо разработать RESTful API на Flask для управления финансовыми подписками пользователя.

Реализовать bash скрипт формирования для формирования changelog из текста коммитов для версии.

Настроить workflow для CI/CD в GitHub Actions.

Требования к Flask приложению:

1. Эндпоинты:

1) Создание подписки: пользователь добавляет новую подписку, указывая название, сумму, периодичность списания (например, ежемесячно, ежегодно) и дату начала.

2) Просмотр подписок: пользователь может видеть все свои активные подписки с деталями.

3) Редактирование подписки: пользователь может обновить информацию о сумме, периодичности или дате следующего списания.

4) Удаление подписки: пользователь может удалить подписку.

Требования к bash скрипту:

1. **Алгоритм работы скрипты**:

1) Проверять наличие последнего тега и собирать все коммиты, сделанные с момента последнего релиза.

2) Генерировать новую секцию в файле changelog.md с указанием версии, даты и списка изменений.

3) Добавлять новые записи для каждого коммита с указанием краткого хэша коммита и ссылки на GitHub для детального просмотра.

2. **Форматирование changelog**:

1) Каждый раздел в changelog.md должен начинаться с заголовка в формате ## [Версия] - Дата, где Версия — это указанный номер версии, например, v1.2.0, а Дата — текущая дата в формате ГГГГ-ММ-ДД.

2) Коммиты внутри каждого раздела должны оформляться списком в формате: Описание коммита [abc123]

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.
2. Проверку безопасности Python-кода с помощью bandit.
3. Запуск скрипта по формированию changelog
4. Публиция changelog на GitHub Pages

Вариант 7

Задание

Реализовать приложение, которое позволяет пользователям создавать заявки на техподдержку, просматривать их статус и обновления, а администраторам — управлять заявками. Система поддерживает роли “пользователь” и “администратор”, где:

1. **Пользователь** может создавать и управлять только своими заявками, а также видеть их статус.
2. **Администратор** имеет полный доступ к управлению всеми заявками.

Настроить workflow для CI/CD в GitHub Actions.

Требования к Flask приложению:

1. Эндпоинты**:**
    1) **POST /register** - регистрация нового пользователя с ролью «пользователь»
    2) **POST /login** - авторизация пользователя для получения доступа к системе.
    3) **POST /tickets** - создание новой заявки на поддержку.
    4) **GET /tickets** - просмотр всех заявок (пользователь видит только свои заявки; администратор видит все заявки).
    5) **GET /tickets/** - просмотр конкретной заявки: (Пользователи видят свои заявки, Администратор видит любые заявки)
    6) **PUT /tickets/** - обновление заявки.
    7) **DELETE /tickets/** - удаление заявки.
    8) **GET /users** - просмотр всех пользователей.
    9) **PUT /users/** - обновление роли пользователя.
2. Авторизация должна выполняться с помощью flask-login

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.

2. Проверку безопасности Python-кода с помощью bandit.

Вариант 8

Задание

Реализовать Flask приложение, которое предоставляет пользователям возможность публиковать статьи, просматривать статьи других авторов и комментировать публикации. Для защиты от несанкционированного доступа в приложение необходимо встроить систему авторизации с ограничением количества попыток входа, предотвращающая брутфорс-атаки. Настроить workflow для CI/CD в GitHub Actions.

Требования к Flask приложению

1. Авторизация должна выполняться с помощью библиотеки Flask Login.
2. Эндпоинты:
    1) **POST /register** — регистрация нового пользователя.
    2) **POST /login** — авторизация пользователя.
    3) **POST /articles** — создание новой статьи.
    4) **GET /articles** — просмотр всех опубликованных статей
    5) **GET /articles/** — просмотр конкретной статьи по её ID
    6) **PUT /articles/** — редактирование собственной статьи (доступно только для её автора).
    7) **DELETE /articles/** — удаление собственной статьи (доступно только для её автора).
    8) **POST /comments** — добавление комментария к статье

Требования к системе защиты

1. Для предотвращения брутфорс-атак ограничить количество:
    1) Не более 5 попыток авторизации для каждого пользователя. При превышении лимита блокировать доступ к эндпоинту /login на 5 минут для конкретного пользователя.
    2) В случае превышения лимита попыток с одного IP-адреса блокировать все попытки входа с этого IP на 15 минут.
2. Возвращать пользователю сообщение о временной блокировке с указанием времени следующей доступной попытки входа.

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.
2. Проверку безопасности Python-кода с помощью bandit.

Инструменты для реализации задания

Для получения ip адреса необходимо запросить поле remote_addr у объекта типа request. Пример:

```
client_ip = request.remote_addr
```

Вариант 9

Задание

Создать RESTful API на Flask для управления запросами к погодной информации с ограничением частоты запросов и кэшированием. Приложение должно ограничивать количество запросов от каждого пользователя и кэшировать результаты, чтобы снизить нагрузку на API и ускорить время отклика для часто запрашиваемых данных. Настроить workflow для CI/CD в GitHub Actions.

Требования к workflow в GitHub Actions

1. **Эндпоинт - GET /weather/** — возвращает текущую информацию о погоде в указанном городе.

    1) Если пользователь запрашивает данные о погоде в городе, для которого информация уже была получена ранее, приложение возвращает кэшированный результат.

    2) Для получения погодных данных достаточно использовать статические заранее подготовленные данные

2. **Кэширование данных:**

    1) Использовать **Flask-Caching** для хранения данных о погоде, чтобы повторные запросы к одному и тому же городу в течение определенного времени (например, 1 часа) возвращали кэшированные данные, а не делали запрос к внешнему API.

    2) Настроить время жизни кэшированных данных (например, 1 час).

3. **Лимитирование запросов:**

    1) Использовать **Flask-Limiter** для ограничения количества запросов к эндпоинту /weather.

2) Ограничить количество запросов с одного IP-адреса (например, не более 10 запросов в течение 1 часа).

3) В случае превышения лимита возвращать сообщение об ошибке с указанием времени, когда пользователь сможет сделать следующий запрос.

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.
2. Проверку безопасности Python-кода с помощью bandit.

Вариант 10

Задание

Разработать RESTful API на Flask, которое предоставляет возможность создавать короткие ссылки, перенаправлять пользователей на исходный URL и отслеживать статистику их использования. Для оптимизации работы необходимо внедрить кэширование (с помощью Flask-Caching) и ограничение количества созданных ссылок на одного пользователя (Flask-Limiter). Настроить workflow для CI/CD в GitHub Actions.

Требования к workflow в GitHub Actions

1. Эндпоинты API:
    1) POST /shorten: Создание короткой ссылки.
        a. Принимает исходный URL и, опционально, пользовательский идентификатор (user_id).
        b. Возвращает уникальный идентификатор для короткой ссылки.
    2) GET /<short_id>: Перенаправление на исходный URL - увеличивает счетчик кликов и регистрирует IP-адрес для статистики.
    3) GET /stats/<short_id>: Просмотр статистики для короткой ссылки. - возвращает количество кликов и список уникальных IP-адресов
2. Кэширование:
    1) Использовать Flask-Caching для кэширования перенаправлений.
    2) Время жизни кэша для перенаправлений — 1 час.
    3) Если ссылка уже кэширована, перенаправление должно происходить без обращения к базе данных.
3. Лимитирование запросов:
    1) Использовать Flask-Limiter для ограничения:
    2) Не более 10 созданий коротких ссылок на одного пользователя в сутки.

3) Не более 100 кликов по одной ссылке с одного IP-адреса в день.

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.
2. Проверку безопасности Python-кода с помощью bandit.

Вариант 11

Задание

Разработать RESTful API на Flask для управления доступом к обучающим материалам на основе атрибутов пользователя, объекта и контекста. Приложение должно учитывать не только роли пользователей, но и их атрибуты, такие как время доступа, статус учетной записи, уровень подписки и т.д. Настроить workflow для CI/CD в GitHub Actions.

Требования к workflow в GitHub Actions

1. Эндпоинты:
    a. POST /register — регистрация нового пользователя с атрибутами:
        i. username (строка),
        ii. password (строка),
        iii. subscription_level (например, basic или premium),
        iv. account_status (active или frozen).
    b. POST /login — аутентификация пользователя.
    c. POST /resources — добавление ресурса с указанием его атрибутов:
        i. name (строка),
        ii. access_level (basic или premium),
        iii. available_hours (диапазон времени, например, 09:00-18:00).
    d. GET /resources — получение списка доступных ресурсов для текущего пользователя.
    e. GET /resources/ — получение конкретного ресурса, если доступ разрешен.
2. Политики доступа (ABAC):
    a. Политики определяются на основе:
    b. Атрибутов пользователя (например, subscription_level, account_status).
    c. Атрибутов ресурса (например, access_level, available_hours).
    d. Контекстных атрибутов (например, текущего времени, IP-адреса).
    e. Примеры политик:

i. Пользователь с premium подпиской имеет доступ ко всем ресурсам.

ii. Пользователь с basic подпиской имеет доступ только к ресурсам с access_level=basic.

f. Доступ ограничен по времени (available_hours).

3. Хранение данных:

a. Использовать базу данных (например, PostgreSQL или SQLite):

b. Таблица users: данные о пользователях (логин, подписка, статус).

c. Таблица resources: данные о ресурсах (название, уровень доступа, доступные часы).

d. Таблица policies: правила доступа (атрибут, оператор, значение).

4. Логика проверки доступа. При запросе к ресурсу приложение проверяет:

a. Все политики для данного ресурса.

b. Совпадение атрибутов пользователя и ресурса с правилами политики.

c. Если хотя бы одно правило не выполняется, доступ запрещается.

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.
2. Проверку безопасности Python-кода с помощью bandit.

Вариант 12

Задание

Разработать RESTful API на Flask для управления запасами товаров (инвентарем) на складе. Приложение должно поддерживать операции по добавлению, обновлению и удалению товаров, а также автоматическую генерацию отчетов о состоянии инвентаря. Настроить workflow для CI/CD в GitHub Actions.

Требования к workflow в GitHub Actions

1. Эндпоинты

1) POST /items — добавление нового товара:

Поля: name (название товара), quantity (количество), price (цена за единицу), category (категория товара).

2) GET /items — получение списка всех товаров с возможностью фильтрации по категории.

3) PUT /items/ — обновление информации о товаре.

4) DELETE /items/ — удаление товара.

5) GET /reports/summary — генерация сводного отчета по текущему состоянию инвентаря:

6) Общая стоимость товаров.

7) Количество товаров в каждой категории.

2. Генерация отчетов:

1) Отчеты формируются автоматически при запросе на /reports/summary.

2) Формат отчетов — JSON и/или CSV.

3) В отчете должно быть указано:

a) Общая стоимость всех товаров.

b) Разбивка по категориям: количество товаров, общая стоимость в категории.

c) Товары с нулевым или отрицательным количеством.

3. Хранение данных

1. Использовать PostgreSQL:

2. Таблица items: данные о товарах (id, name, quantity, price, category).

4. Функциональность

1. Проверка корректности данных при добавлении/обновлении товаров:

2. Количество не может быть отрицательным.

3. Цена должна быть больше нуля.

Требования к workflow в GitHub Actions

1. Автоматический запуск unit-тестов при каждом push в ветку main.
2. Проверку безопасности Python-кода с помощью bandit.

Оформление

Объем работы от 8 страниц. Ориентация листа – книжная, размер – А4, формат полей – «обычные». Выравнивание – по ширине, отступ первой строки – 1,25, межстрочный интервал – 1,5, отсутствие интервалов до и после абзацев.

Шрифт – Times New Roman, кегль – 14 пт, цвет текста – черный. Содержание должно быть автособираемое.

Название главы выравнивается по центру без абзацного отступа. Размер кегля 16 пт или 14 пт. Шрифт полужирный.

Название подраздела выравнивается по левому краю, без абзацного отступа. Кегль 14 пт. Шрифт полужирный.

Номера страниц — 14 пт. Расположение — справа внизу. Первый лист (титульный) не нумеруется. Лист с содержанием должен иметь номер 2.

Нумерация рисунков возможна либо по порядку (Рисунок 1, Рисунок 2 и т. п.), либо по разделам (Рисунок 1.1, Рисунок 2.2).

Рисунок и подпись к рисунку выравниваются по центру без абзацного отступа.

Инструкция по установке Python библиотек

1. pip install flasgger
2. pip install flask
3. pip install bandit
4. pip install flask-login

Структура отчёта

Структура отчёта должна быть следующая:

1. Титульный лист;
2. Содержание;
3. Введение;
4. Глава 1;
5. Глава 2;
6. Заключение;
7. Список использованных источников;
8. Приложение(-я).

Первая глава представляет из себя описание используемых технологий, а также, если использовались, библиотек, фреймворков.

Вторая глава. В данном разделе необходимо описать выполненную работу — описать вычисление варианта и используемую функцию для вычисления показателя доходности ценной бумаги.

Список использованных источников. Минимум 3 источника. Оформление в соответствии с ГОСТом 2021 года.

Приложения. В приложении помещаются: исходный код.