Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего образования

**«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)**

Факультет дистанционного обучения (ФДО)

А. Я. Суханов

РАЗРАБОТКА ВЕБ-СЕРВИСОВ ДЛЯ НАУЧНЫХ И ПРИКЛАДНЫХ ЗАДАЧ

**Методические указания
по выполнению лабораторных работ
для студентов, обучающихся с применением
дистанционных образовательных технологий
(дополнительное издание)**

Томск 2023

Корректор: А. Н. Миронова

Суханов А. Я.
Разработка веб-сервисов для научных и прикладных задач : методические указания по выполнению лабораторных работ для студентов, обучающихся с применением дистанционных образовательных технологий (доп. издание) / А. Я. Суханов. – Томск : ФДО, ТУСУР, 2023. – 40 с.

© Суханов А. Я., 2023
© Оформление.
ФДО, ТУСУР, 2023

ОГЛАВЛЕНИЕ

Введение **4**

Изучение фреймворка FastAPI и процесса деплоя **5**

1 Пример использования в качестве системы контроля версий GitFlic 5

2 Создание проекта в PyCharm с использованием Pipenv 8

3 Push проекта на GitHub и тестирование на GitActions 13

4 Запуск uvicorn и работа с веб-приложением 20

5 Deploy на render.com 22

6 Продолжение реализации проекта. Создание шаблона FastAPI 27

7 Добавление работы с формами 34

8 Примеры API функций 39

Полезные ресурсы **40**

ВВЕДЕНИЕ

Настоящие дополнительные методические указания предназначены для выполнения лабораторных работ по дисциплине «Разработка веб-сервисов для научных и прикладных задач» и разработаны с учетом требований ФГОС ВО.

Цель лабораторных работ: получение навыков разработки веб-приложений и веб-сервисов с использованием одного из фреймворков и типовых научных библиотек для решения простейших задач по обработке данных.

Дополнительное издание описывает основные этапы работы на примере GitHub Actions (GitActions), render, Pipenv и FastAPI.

ИЗУЧЕНИЕ ФРЕЙМВОРКА FASTAPI И ПРОЦЕССА ДЕПЛОЯ

Все указания приводятся для версии Python 3.8.10. Следует установить Python версии 3.8 и PyCharm. Если вы пользуетесь другими версиями Python, то постарайтесь самостоятельно исправлять проблемы, связанные с несоответствием версий библиотек и установкой нужного программного обеспечения. Развитие данного умения также является основополагающим моментом в обучении и приобретении необходимых навыков. Не возбраняется пользоваться удаленными ресурсами и поисковиками для устранения проблем и возможных ошибок.

1 Пример использования в качестве системы контроля версий GitFlic

Кратко рассмотрим российский аналог систем типа GitHub. К сожалению, таких возможностей, как GitHub, он не предоставляет, тем не менее, по заявлениям разработчиков, в нем присутствует и система CI (онлайн – в платном варианте, бесплатно ее можно развернуть на локальной машине пользователя). Далее будем рассматривать систему GitHub, но здесь приведем начальный пример работы с GitFlic. Можно начать выполнение лабораторной работы с этого момента, так как здесь будут приведены примеры взаимодействия PyCharm и системы контроля версий Git.

Регистрируемся на gitflic.ru. Вводим пароль и свою почту, куда придет письмо для подтверждения аккаунта. Создаем проект, выбрав язык программирования, и добавляем название проекта в поле ввода (рис. 1.1).

Установите Git:

sudo apt install git

Проверьте, что он установился:

git -version

Язык программирования

Python

URL проекта

https://gitflic.ru/project/sayc/ fastlab

Описание

Описание

Публичный проект

Любой пользователь интернета может увидеть этот репозиторий. Вы сами выбираете кто сможет делать коммиты в этот репозиторий. Обратитесь к администратору сервиса для получения доступа.

Приватный проект

Вы сами выбираете кто может увидеть этот репозиторий и кто сможет делать в нем коммиты

Создать проект

Рис. 1.1 – Пример создания проекта на GitFlic

Далее приведены примеры работы с системой окружения Pipenv (в основных методических указаниях приведен пример работы с virtualenv). Вы можете использовать либо ту, либо другую систему.

Краткая информация по работе с Pipenv

В совместной разработке на Python лучше использовать Pipenv, которая позволяет управлять окружениями enviroments, устанавливать и обновлять пакеты (pip), используя файл Pipfile.lock, устанавливать детерминированный набор пакетов. Например, Pipenv удобно применять, если разные пакеты используются в зависимости от пакетов разных версий.

Используются два файла: Pipfile, являющийся фактически заменой requirements.txt, и Pipfile.lock, в котором находятся все зависимости, отслеживающиеся автоматически (дерево зависимостей).

Для создания или инициализации среды можно использовать команду:

pipenv shell,

если среды не существует, то она будет создана.

В текущей папке будет создан файл Pipfile, а среда будет создана в папке по умолчанию (например, /home/user/.local/share/virtualenvs/…), если запуск произошел из командной строки и не из environment. Если же запуск произошел из environment, то будет инициализирована среда, соответствующая текущей папке.

Перед запуском нужно перейти в папку вашего проекта. Если же используется PyCharm, то при выборе папки проекта установка среды и создание файла Pipfile автоматически реализуется в ней, среда размещается в папке по умолчанию для данной операционной системы. Удалить среду можно с помощью команды pipenv -rm, запустив ее в папке, где находится файл Pipefile или откуда вызывалась команда pipenv shell.

Генерация Pipefile.lock будет выполнена командой:

pipenv lock

Установка всех пакетов в окружении из Pipefile.lock:

pipenv sync

Удаление всех пакетов, установленных для данного environment, которых нет в Pipefile.lock:

pipenv clean

Для установки пакетов используется команда:

pipenv install имя_пакета

Вызов команды

pipenv install

произведет установку всех пакетов, указанных в Pipfile, и приведет к обновлению Pipfile.lock.

2 Создание проекта в PyCharm с использованием Pipenv

Создадим проект в PyCharm.

Способ выбора Pipenv в PyCharm представлен на рисунке 1.2.

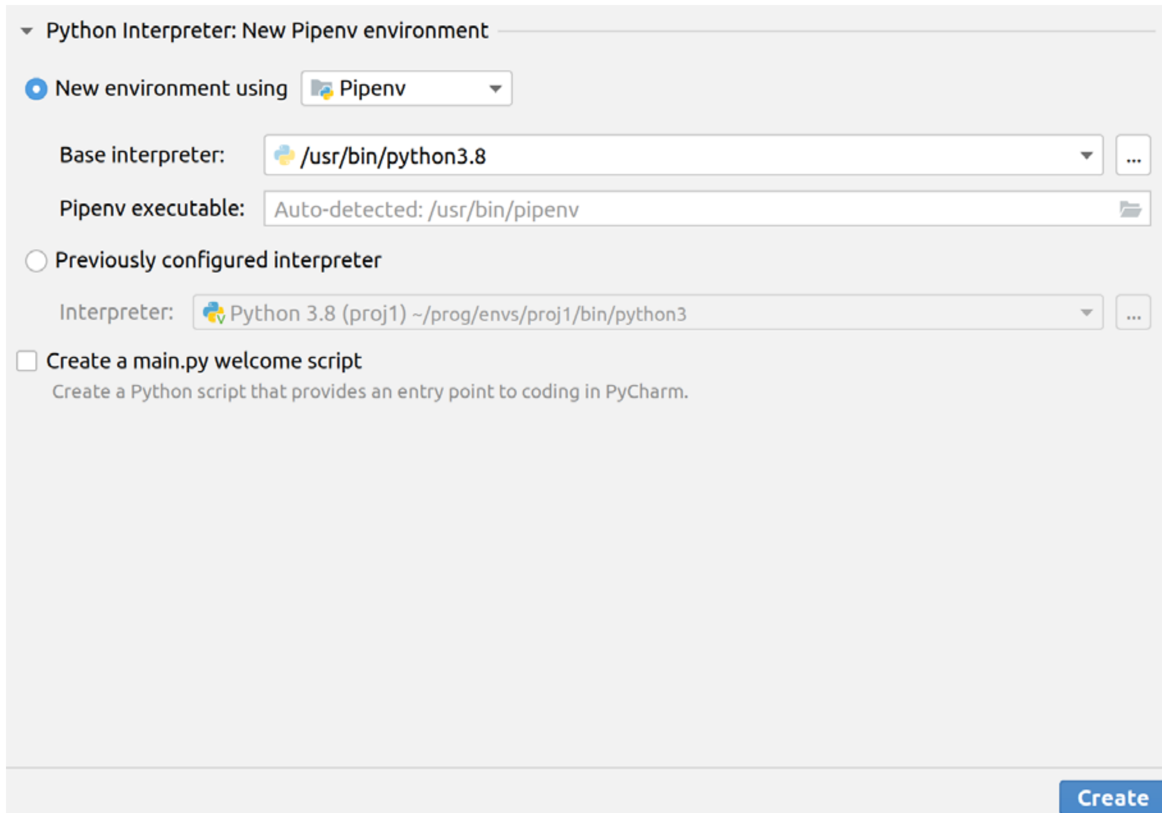

Рис. 1.2 – Выбор Pipenv

Подключение git в PyCharm и push на Gitflic

Далее подключим Git. Выберем вкладу VCS → Enable version control integration.

Создадим файл .gitignore, при добавлении в Git нажимаем Cancel, чтобы не добавлять его в отслеживание системой контроля версий, в этом файле указываем строки (иногда, наоборот, лучше добавить pipfile.lock в отслеживаемые, чтобы потом не запускать команды создания данного файла):

.idea/.

Pipfile.lock

Правой кнопкой мышки щелкаем на Pipfile, выбираем GIT и add, тем самым добавляя данный файл в отслеживаемые, в результате он должен стать зеленого цвета.

Далее можно сделать Commit во вкладке GIT.

Commit сохранит отслеживаемые файлы в системе контроля версий, push позволяет их поместить на удаленный сервер, pull – взять на ваш локальный компьютер с сервера.

В папке нашего проекта запускаем через терминал команды, указанные в проекте GitFlic.

```
git config --global user.name "*****"
git config --global user.email "***@some_your_mail.org"
```

Пример окна с Commit в PyCharm приведен на рисунке 1.3.

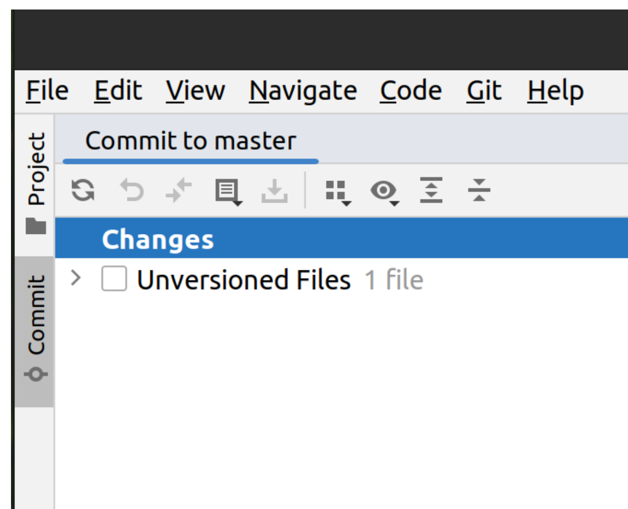

Рис. 1.3 – Пример Commit в PyCharm

Можно переключаться между Commit и Project.

Создание начального проекта и push на GitFlic

Разработаем веб-сервис, используя фреймворк FastAPI. Данный фреймворк позволяет разрабатывать приложения довольно быстро, в том

числе асинхронные. FastAPI базируется на двух фреймворках: Pydantic и Starlette, первый отвечает за валидацию данных, второй – за работу с Вебом. Также FastAPI поддерживает REST API, документацию Swagger и авторизацию OAuth *2.0*.

Для установки нужных библиотек можно изменить GitFlic:

```
[packages]
fastapi = "*"
pydantic = "*"
uvicorn = "*"
```

Также для поиска информации об управлении пакетами можно обратиться к инструкции по использованию Pipfile:

https://www.jetbrains.com/help/pycharm/using-pipfile.html

Можно в терминале вызывать команду pipenv update или сделать запрос на обновление и установку зависимостей в PyCharm (рис. 1.4).

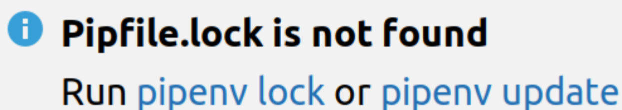

Рис. 1.4 – Пример запроса на обновление
и установку зависимостей в PyCharm

Можно записать изменения в pipenv.lock-файл или записать и установить все пакеты и зависимости с помощью команды pipenv update.

Создадим в нашем проекте файл fastlab.py с кодом. Пример на FastAPI.

```
from fastapi import FastAPI
import uvicorn
app = FastAPI()
# Hello World route
```

```
@app.get("/")
def read_root():
    return {"Hello": "World"}
if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Документацию можно посмотреть здесь:

http://0.0.0.0:8000/docs

Запустим в PyCharm данный файл и посмотрим, как работает наше приложение, перейдя по предложенной ссылке.

Конечно, запускать uvicorn можно не из самого файла Python, а наоборот, использовать uvicorn для запуска приложения.

Создадим файл README.md. Запишем туда нужную нам информацию касательно проекта. Можно откатить добавленные для отслеживания файлы с помощью GIT → Rollback.

В .gitignore можно добавить Pipefile.lock, но можно и не добавлять, дальнейшие указания предполагают, что мы не отслеживаем данный файл, и тогда его надо создавать при запуске тестов и при деплое (deploy).

Итоговое содержимое нашего проекта (рис. 1.5):

```
fastlab/
 .gitignore
 fastlab.py
 Pipfile
 Pipfile.lock
 README.md
```

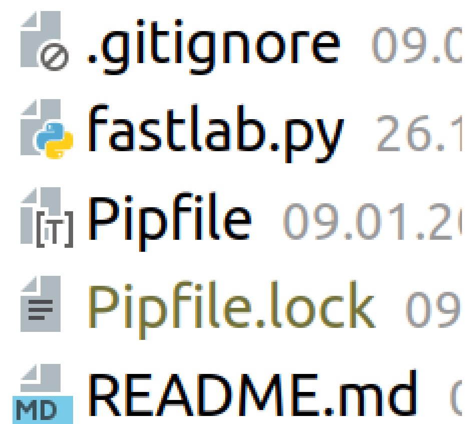

Рис. 1.5 – Итоговое содержимое проекта

Теперь сделаем push на наш репозиторий в GitFlic. Необходимо ввести пароль и адрес указываемой командой:

git remote add origin https://gitflic.ru/project/ваш логин/имя проекта.git

При создании проекта этот адрес указывается снизу для помощи в пункте «Быстрая настройка», если вы уже делали подобное раньше. Следует «запушить» существующий репозиторий.

Для работы с CI/CD можно использовать standalone-версию. Можете рассмотреть этот вариант самостоятельно.

3 Push проекта на GitHub и тестирование на GitActions

GitHub является известной системой контроля версий, кроме того, достаточно большой набор функций там предоставляется бесплатно.

Рассмотрим вариант использования GitHub и GitActions. Зарегистрируемся на github.com и создадим новый репозиторий (рис. 1.6).

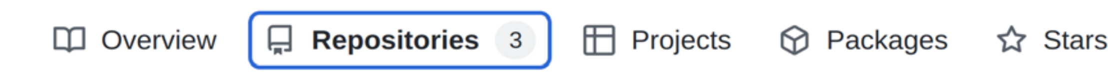

Рис. 1.6 – Выбор или создание репозитория на GitHub

Выберем *Репозиторий* и далее *Создание нового репозитория* (зеленая кнопка *New*).

Затем укажем имя репозитория, которое потом будем использовать (рис. 1.7).

Repository name *

Рис. 1.7 – Ввод имени репозитория

Бесплатно можно использовать как публичный, так и закрытый репозиторий.

После создания репозитория уже готовый проект из PyCharm «запушим» на GitHub, указав адрес, который можно получить во вкладке *Code* на GitHub. Также далее можно указать специальный токен доступа либо напрямую логин/пароль. Чтобы получить токен, можно проделать следующие шаги:

1. Нажмите на свой аватар в правом верхнем углу и перейдите в меню *Settings*.

2. Из боковой левой панели перейдите в *Developer settings > Personal access tokens* (в самом низу). Нажмите *Generate new token*, при необходимости введите пароль учетной записи.

3. В поле *Note* введите назначение токена.

Выбрать token(classic) и галочки Repo, workflow, read.org, gist.

Далее пробуем во вкладке PyCharm или в командной строке запустить команду git push в ветку main, указав во всплывающем окне адрес HTTPS и далее сгенерированный токен.

В случае если push не сработает (из-за несовпадения истории commit, например, если в удаленный репозиторий была добавлена лицензия), может потребоваться слияние проектов. Поэтому сначала делаем pull, при этом переименовывая ветку в main или давая то название, которое есть в GitHub. Указываем параметр rebase во вкладке *Modify options* (рис. 1.8). Rebase – это наложение commit'ов поверх другого базового commit'a. Под базовым понимается тот commit, к которому применяются commit'ы выбранной ветки.

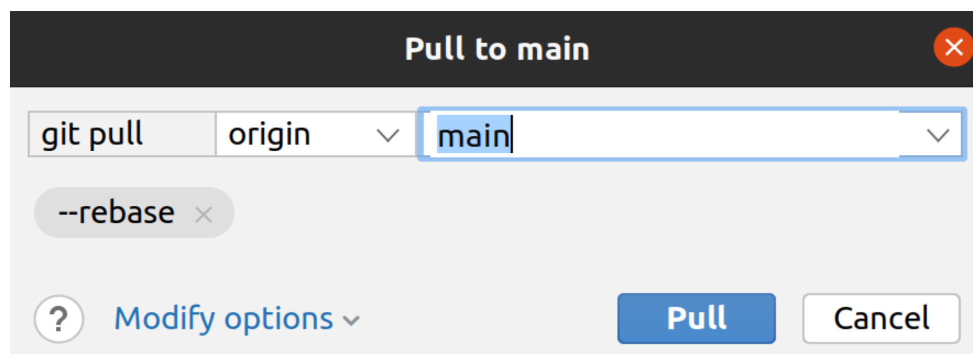

Рис. 1.8 – Пример выполнения pull с git

Можно посмотреть разницу двух веток: удаленной и локальной (рис. 1.9). Далее снова делаем push.

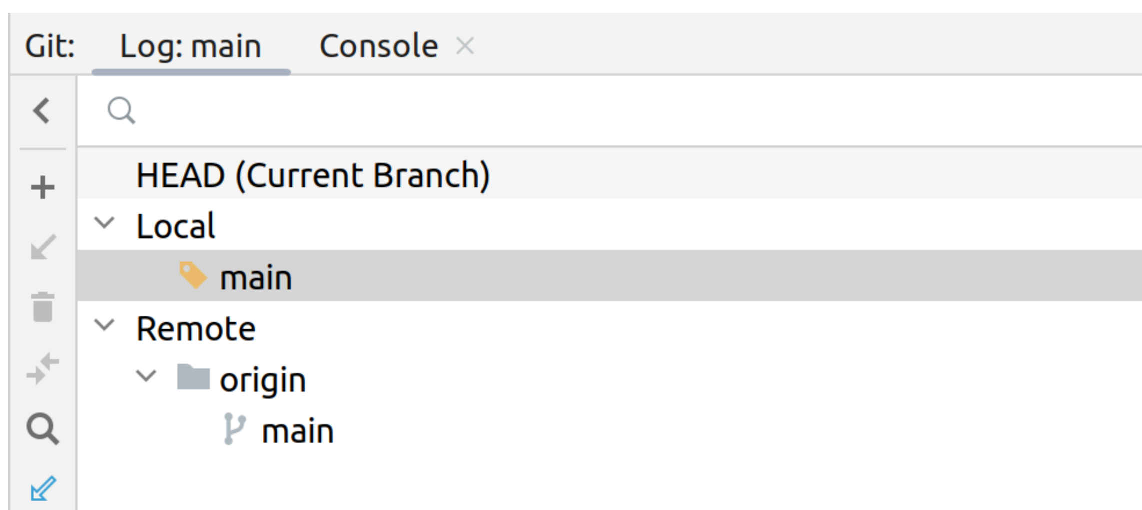

Рис. 1.9 – Просмотр разницы двух веток в PyCharm

Тестирование с помощью workflows

В нашем проекте PyCharm создадим следующие папки:

.github

workflows

my.yaml

В my.yaml-файл, который будет использоваться для запуска actions на GitHub на событие push, добавим следующие декларации:

```
# имя, отображаемое в интерфейсе
name: 'test my project'
on: [push] # список событий, на которые запускается действие
jobs: # список работ, которые будут производиться (каждая работа будет выводиться отдельно)
  checks: # имя работы checks
    runs-on: ubuntu-latest # на какой машине делать работы (можно задать матрицы машин, допустимы Windows и Mac OS)
    steps: # выполняемые последовательно шаги
      - uses: actions/checkout@v3
      - run: echo "hello world"
```

Ключевое слово uses указывает на то, что на этом шаге будет запущена версия 3 «действия»/«действия проверки». Это действие, которое проверяет ваш репозиторий на исполнителе, позволяя вам запускать сценарии или другие действия с вашим кодом (например, инструменты сборки и тестирования). Вы должны использовать «действие проверки» каждый раз, когда ваш рабочий процесс будет выполняться с кодом репозитория.

Run запускает команду, в данном случае команду echo.

Теперь попробуем протестировать наше приложение с помощью GitActions, и если тесты не пройдут успешно, то commit на GitHub не сработает. Дальнейшей целью будет сделать deploy на удаленную PaaS-систему. Вы можете выбрать любую, которая вам доступна, естественно, указания для них будут отличаться от предложенного здесь варианта render.

Изменим наш yaml-файл, добавив следующее содержимое в соответствующий блок, и сделаем еще один commit и push, данные commit'ы позволят установить Python и Pipenv.

```
checks:
  runs-on: ubuntu-latest
  steps:
    - name: Begin
      uses: actions/checkout@v3
    - name: Echo hello
      run: echo "hello world"

    - name: Setup Python
      uses: actions/setup-python@v2
      with:
        python-version: "3.8"

    - name: Install pipenv
      run: python -m pip install pipenv
```

Все изменения на GitHub можно смотреть во вкладке actions. Для этого необходимо щелкнуть мышкой на соответствующем commit'e и посмотреть workflow check.

Теперь нужно установить требуемое окружение и запустить наше приложение или тесты.

Содержимое Pipefile

```
[packages]
fastapi = "*"
pydantic = "*"
uvicorn = "*"
pytest = "*"
[dev-packages]
[requires]
python_version = "3.8"
```

Добавим в yaml-файл установку требуемых зависимостей и запуск в среде окружения нового файла с тестом.

```
- name: Pipenvlock
  run: pipenv lock
- name: Pipenvsync
  run: |
    pipenv sync
- name: Start tests
  run: pipenv run python -m pytest my_tests.py
```

Последняя строчка запускает тестирование в заданном окружении Pipenv с учетом установленных библиотек.

Добавим файл с тестом, в котором проверяется, что 2 + 2 = 4.

Файл my_tests.py.

```
import fastlab
```

```
def test1():
   assert  fastlab.sum_two_args(2,2) == 4
```

Изменим файл fastlab.py, добавив функцию, которая суммирует два аргумента.

```
import uvicorn
app = FastAPI()

def sum_two_args(x,y):
   return x+y
# Hello World route
@app.get("/")
def read_root():
   return {"Hello": "World"}
if __name__ == "__main__":
   uvicorn.run(app, host="0.0.0.0", port=8000)
```

В данных примерах мы добавляем одну функцию в наш файл, и затем тестируем ее в модуле тестирования, это так называемое unit-тестирование.

Проведем изменения, добавим my_tests.py в отслеживаемые git, сделаем commit и push. Не забывайте указывать в commit'e, какие изменения вы делаете. Смотрим результаты на GitHub. Не переживайте, если сразу что-то не получилось, можно посмотреть ошибки и попытаться их исправить, не бойтесь пользоваться поисковиком.

Добавим заведомо два ошибочных теста в файл my_tests.py и один верный и посмотрим вывод, который будет подготовлен в CI/CD после push.

```
def test2():
   assert  fastlab.sum_two_args(2.0001,2) == 4
def test3():
```

```
    assert  fastlab.sum_two_args(3,2) == 5
def test4():
    assert  fastlab.sum_two_args("2",2) == 4
```

Очевидно, что проверка не дала положительного результата, можно посмотреть ошибки и исправить их.

```
def test2():
    assert  fastlab.sum_two_args(2.0001,2) == 4.0001
def test3():
    assert  fastlab.sum_two_args(3,2) == 5
def test4():
    assert  fastlab.sum_two_args("2","2") == "22"
```

Снова запустим. Тесты должны пройти.

4 Запуск uvicorn и работа с веб-приложением

Дальше рассмотрим работу с нашим веб-приложением.

Для запуска uvicorn используем его возможности.

Запустить веб-приложение Python:

uvicorn {{import.path:app_object}}

- Запустить сервер на порте 8080 на локальном хосте (localhost):

uvicorn --host {{localhost}} --port {{8080}} {{import.path:app_object}}

- Включить «горячую перезагрузку» для отслеживания изменений:

uvicorn --reload {{import.path:app_object}}

- Использовать 4 рабочих процесса для обработки запросов:

uvicorn --workers {{4}} {{import.path:app_object}}

- Запустить приложение по HTTPS:

uvicorn --ssl-certfile {{cert.pem}} --ssl-keyfile {{key.pem}} {{import.path:app_object}}

В нашем случае используем команду:

uvicorn --host 127.0.0.1 --port 8000 fastlab:app

Запустим ее в терминале PyCharm, предварительно удалив из файла fastlab запуск uvicorn.

```
app = FastAPI()
def sum_two_args(x,y):
   return x+y
# Hello World route
@app.get("/")
def read_root():
   return {"Hello": "World"}
```

Обычно тесты подобных веб-приложений можно реализовать с помощью такого средства, как selenium, но, к сожалению, при его выборе могут возникнуть трудности при установке и дальнейшей работе с ним, поэтому в рамках лабораторной работы лишь приведем начальный код, который поможет вам запустить работу с selenium на Python. Вы можете самостоятельно добавить в свой код любые тесты на selenium, изучив материалы в Интернете.

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service as ChromeService
from webdriver_manager.chrome import ChromeDriverManager
options = webdriver.ChromeOptions()
# отключает режим показа браузера
#options.add_argument('--headless')
# пример других опций, которые вы можете использовать
#options.add_argument('--disable-gpu')
#options.add_argument("--disable-popup-blocking")
# создать драйвер браузера Chromium
driver = webdriver.Chrome(options=options, ser-
vice=ChromeService(ChromeDriverManager().install()))
# получить страницу
driver.get("http://127.0.0.1:8000Ошибка! Недопустимый объект гиперс-
сылки.)
print(driver.title)
driver.close()
```

5 Deploy на render.com

Далее попробуем сделать deploy нашего проекта на render.com. Для этого необходимо зарегистрироваться на ресурсе любым доступным образом. Так как вы уже зарегистрированы на GitHub, то можете зайти туда под аккаутном GitHub. Далее выбираем тарифный план (среди них есть и бесплатный).

Далее на вкладке dashboard выбираем, например, веб-сервисы (рис. 1.10).

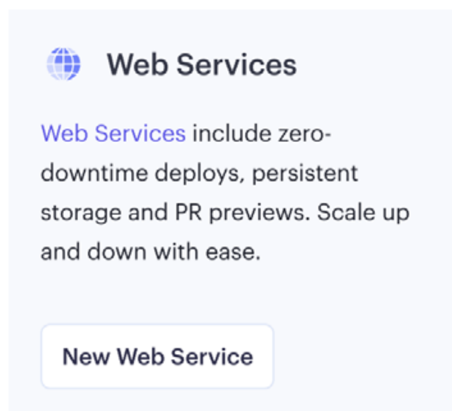

Рис. 1.10 – Создание веб-сервиса на render.com

Настраиваем connect с GitHub справа или выбираем ваш репозиторий, если он доступен (рис. 1.11).

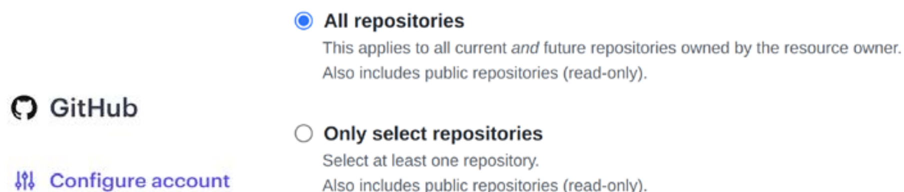

Рис. 1.11 – Создание коннекта с репозиторием на GitHub

Можно выбрать все репозитории или какой-то конкретный, можно указать ваш для лабораторной работы.

Затем вводите пароль для входа в GitHub.

Перед тем как нажать кнопку *Connect* на render.com, необходимо **добавить Pipfile.lock в отслеживаемые git.**

Изменим наш yaml-файл на следующее содержимое:

```
name: 'test my project'
on: [push]
jobs:
  checks:
    runs-on: ubuntu-latest
    steps:
      - name: Begin
        uses: actions/checkout@v3
      - name: Echo hello
        run: echo "hello world"

      - name: Setup Python
        uses: actions/setup-python@v2
        with:
          python-version: "3.8"

      - name: Install pipenv
        run: python -m pip install pipenv

      - name: Pipenvsync
        run: |
          pipenv sync
      - name: Start tests
        run: pipenv run python -m pytest my_tests.py
```

Файл fastlab.py:

```
from fastapi import FastAPI
```

```
app = FastAPI()
def sum_two_args(x,y):
   return x+y
# Hello World route
@app.get("/")
def read_root():
   return {"Hello": "World"}
```

После этого нажимаем кнопку *Connect* на выбранном репозитории на render.com.

Указываем серверы, где хотим разместить наш сервис, его имя и команды для запуска (рис. 1.12, 1.13).

```
python -m pip install pipenv && pipenv sync
```

Build Command

This command runs in the root directory of your repository when a new version of your code is pushed, or when you deploy manually. It is typically a script that installs libraries, runs migrations, or compiles resources needed by your app.

```
$ python -m pip install pipenv && pipenv sync
```

Рис. 1.12 – Настройка build приложения перед его запуском

```
pipenv run python -m uvicorn fastlab:app --host 0.0.0.0
```

Start Command

This command runs in the root directory of your app and is responsible for starting its processes. It is typically used to start a webserver for your app. It can access environment variables defined by you in Render.

```
$ pipenv run python -m uvicorn fastlab:app --host 0.0.0.0
```

Рис. 1.13 – Настройка запуска самого приложения

После достаточно длительного процесса наш мини-веб-сервис готов (рис. 1.14).

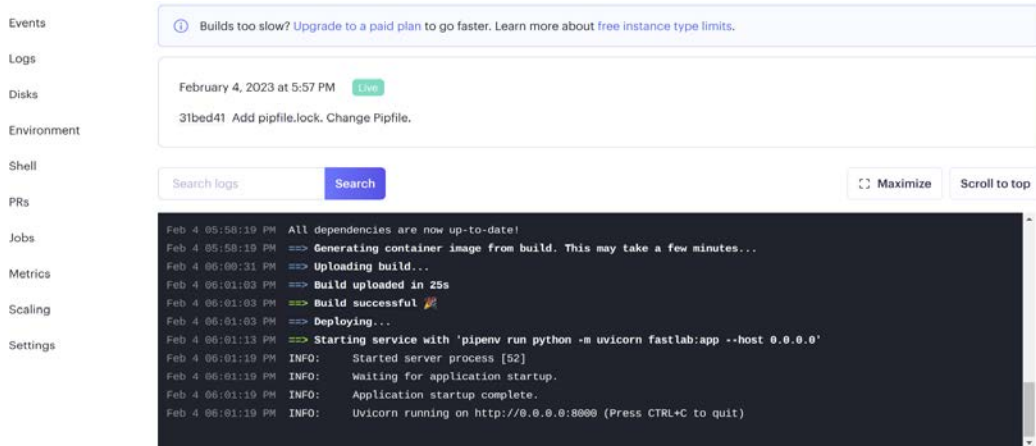

Рис. 1.14 – Просмотр результата build и запуска приложения в окне на render.com

Теперь после каждого push на GitHub будет проходить процесс деплоя на render.com.

Указанное имя сервиса доступно по приведенной ссылке: https://some_name.onrender.com. *some_name* – это имя, которое вы указали.

Также вы можете попытаться использовать другие PaaS-системы, которые вам доступны, обычно они платные.

Процесс деплоя виден во вкладке events, там же можно нажать на Deploy и посмотреть весь процесс, кроме того, доступны логи выполнения вашей программы и ее вывод (рис. 1.15).

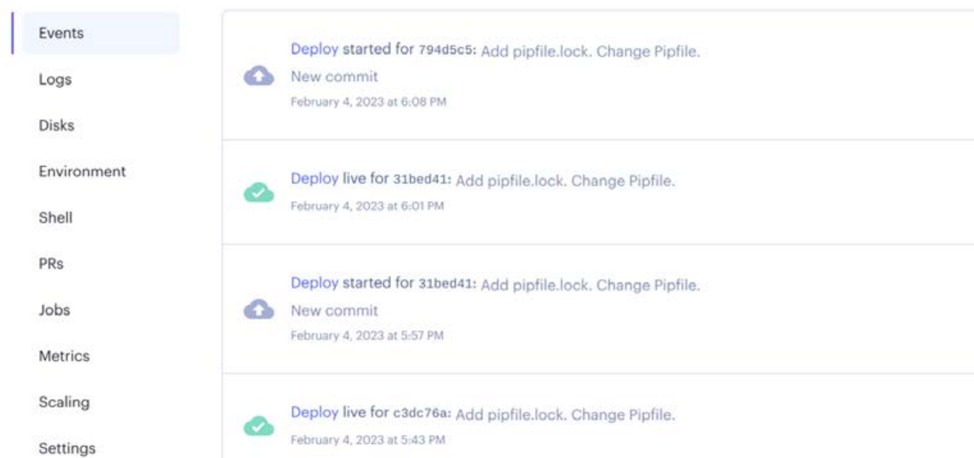

Рис. 1.15 – Отображения событий на render.com при операциях push и deploy

Вообще в дальнейшем можно не делать push, если об этом не сказано отдельно, если это занимает время или не хочется ждать результатов. Можно просто смотреть все добавления и изменения непосредственно на локальной машине.

Можете использовать команду для запуска вашего приложения из терминала PyCharm.

```
uvicorn --host 127.0.0.1 --port  8000 fastlab:app
```

6 Продолжение реализации проекта. Создание шаблона FastAPI

Здесь мы реализуем шаблоны для вывода созданной картинки, как сохраненной в static, так и выдаваемой через url и запрос get.

Создадим каталоги static и templates, в каталог static будем класть css-файлы, статические изображения и т. д., в templates – шаблоны html-файлов, которые будут рендериться (обрабатываться) с помощью jinja и далее в виде html отдаваться на сторону клиента (типичная работа динамического веб-сервера, запустить скрипт, который сформирует страницу).

В каталоге templates создадим файл some.html. Автоматически PyCharm добавит туда теги.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <link href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet">
</head>
<body>
    <h1>Something from user : {{ something }}</h1>
</body>
</html>
```

В каталог static добавим файл styles.css, который задаст цвет заголовка h1 в html-файле.

```
h1 {
    color: blue;
}
```

В Pipefile добавим строчку jinja2 = "*".

Изменим код файла fastlab.py на следующий:

```
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
app = FastAPI()
def sum_two_args(x,y):
    return x+y
# Hello World route
@app.get("/")
def read_root():
    return {"Hello": "World"}
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")
@app.get("/some_url/{something}", response_class=HTMLResponse)
async def read_something(request: Request, something: str):
    return templates.TemplateResponse("some.html", {"request": request, "something": something})
```

Запустим из терминала:

```
uvicorn --host 127.0.0.1 --port  8000 fastlab:app
```

В строке браузера введем адрес:

127.0.0.1:8000/some_url/12345_string

В результате должна появиться надпись, которая была в some.html, и значение, которое введено после some_url.

Важное замечание: если вы собираетесь далее делать deploy на сервер render, то задайте нужную версию Python в Environments. Например, 3.8.10.

Переменная PYTHON_VERSION (рис. 1.16). В поле ввода текста (Value) указываем версию, например 3.8.10, если не задать эту переменную, то по умолчанию на render будет версия Python 3.7.

Рис. 16 – Изменение версии Python на 3.8

Теперь сделаем вывод картинки.

Создадим еще один файл с шаблоном html (image.html) и поместим его в templates. Можете назвать его по-другому, если захотите.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Images</title>
   <link href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet">
</head>
<body>
<h1> Images </h1>
 <table>
  <tr>
   <td> Изображение static </td>
   <td> Изображение из буфера </td>
  </tr>
```

```
  <tr>
    <td> <img src = "{{ im_st }}"/>  </td>
    <td> <img src = "{{ im_dyn }}"/> </td>
  </tr>
</table>
</body>
</html>
```

Как видите, в файле есть две переменные в фигурных скобках. То, что находится в этих скобках, воспринимается интерпретатором как выражение.

В том числе url_for есть выражение, вызывающее функцию, которая у нас появится и в файле fastlab.py.

В css-файл style.css добавьте строки, чтобы у таблицы были границы:

```
table, th, td {
  border: 1px solid black;
}
```

В Pipfile добавьте строки:

```
jinja2 = "*"
numpy = "*"
Pillow = "*"
```

Далее укажем, что теперь у нас будет в файле fastlab.py:

```
import fastapi.responses
import numpy
import io
from PIL import Image
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
```

```
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
app = FastAPI()
def sum_two_args(x,y):
    return x+y
# Hello World route
@app.get("/")
def read_root():
    return {"Hello": "World"}
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")
# возвращаем some.html, сгенерированный из шаблона
# передав туда одно значение something
@app.get("/some_url/{something}", response_class=HTMLResponse)
async def read_something(request: Request, something: str):
    return  templates.TemplateResponse("some.html",  {"request":  request,
"something": something})
def create_some_image(some_difs):
    imx = 200
    imy = 200
    image = numpy.zeros((imx,imy, 3), dtype=numpy.int8)
    image[0:imy//2,0:imx//2,0] = some_difs
    image[imy//2:,imx//2:,2] = 240
    image[imy//2:,0:imx//2, 1] = 240
    return image
# возврат изображения в виде потока медиаданных по URL
@app.get("/bimage", response_class=fastapi.responses.StreamingResponse)
async def b_image(request: Request):
    # рисуем изображение, сюда можете вставить GAN, WGAN сети и т. д.
```

```
    # взять изображение из массива в Image PIL
    image = create_some_image(100)
    im = Image.fromarray(image, mode="RGB")
    # сохраняем изображение в буфере оперативной памяти
    imgio = io.BytesIO()
    im.save(imgio, 'JPEG')
    imgio.seek(0)
    # Возвращаем изображение в виде mime типа image/jpeg
    return fastapi.responses.StreamingResponse(content=imgio, media_type="image/jpeg")
# возврат двух изображений в таблице html, одна ячейка ссылается на url bimage
# другая ячейка указывает на файл из папки static по ссылке
# при этом файл туда предварительно сохраняется после генерации из массива
@app.get("/image", response_class=HTMLResponse)
async def make_image(request: Request):
    image_n = "image.jpg"
    image_dyn = request.base_url.path+"bimage"
    image_st = request.url_for("static", path = f'/{image_n}')
    image = create_some_image(250)
    im = Image.fromarray(image, mode="RGB")
    im.save(f"./static/{image_n}")
    # передаем в шаблон две переменные, к которым сохранили url
    return templates.TemplateResponse("image.html", {"request": request, "im_st":image_st, "im_dyn": image_dyn})
```

Можно теперь опять запустить наш проект из терминала.

uvicorn --host 127.0.0.1 --port 8000 fastlab:app

Далее открываем по ссылке браузер и вводим адреса:

127.0.0.1:8000/image

127.0.0.1:8000/bimage

Смотрим, что получилось. Далее можно сделать commit и deploy, посмотреть на результат. Если возникают проблемы, нужно проверить совместимость библиотек и версии Python (здесь 3.8.10). Установите переменную Environment на render.com PYTHON_VERSION 3.8.10 либо вашу версию Python.

7 Добавление работы с формами

Теперь рассмотрим работу с формами на FastAPI. Сделаем простейшую возможность открытия файлов с изображениями, их изменения и вывода на странице. Здесь уже появится выполнение post-запроса и передача данных формы через multipart-формат.

Добавим в файл fastapi следующий код, содержащий обработку get и post-запросы с формой.

```
from fastapi import Form, File, UploadFile
from typing import List
import hashlib
from PIL import ImageDraw

@app.post("/image_form", response_class=HTMLResponse)
async def make_image(request: Request,
                name_op:str = Form(),
                number_op:int = Form(),
                r:int = Form(),
                g:int = Form(),
                b:int = Form(),
                files: List[UploadFile] = File(description="Multiple files as UploadFile")
                ):
    # устанавливаем готовность прорисовки файлов, можно здесь проверить, что файлы вообще есть
    # лучше использовать исключения
    ready = False
    print(len(files))
    if(len(files)>0):
        if(len(files[0].filename)>0):
```

```
            ready = True
        images = []
        if ready:
          print([file.filename.encode('utf-8') for file in files])
          #  преобразуем имена файлов в хеш -строку
          images  =  ["static/"+hashlib.sha256(file.filename.encode('utf-8')).hexdigest()
for file in files]
          # берем содержимое файлов
          content = [await file.read() for file in files]
          # создаем объекты Image типа RGB размером 200 на 200
          p_images  =  [Image.open(io.BytesIO(con)).convert("RGB").resize((200,200))
for con in content]
          # сохраняем изображения в папке static
          for i in range(len(p_images)):
            draw = ImageDraw.Draw(p_images[i])
            # Рисуем красный эллипс с черной окантовкой
            draw.ellipse((100, 100, 150, 200+number_op), fill=(r,g,b), outline=(0, 0, 0))
            p_images[i].save("./"+images[i],'JPEG')
        # возвращаем html с параметрами-ссылками на изображения, кото-
рые позже будут
        # извлечены браузером запросами get по указанным ссылкам в img
src
        return  templates.TemplateResponse("forms.html",  {"request":  request,
"ready": ready, "images": images})

    @app.get("/image_form", response_class=HTMLResponse)
    async def make_image(request: Request):
        return templates.TemplateResponse("forms.html", {"request": request})
```

Добавим код шаблона с формой, в этом шаблоне происходит проверка готовности изображений и вывод их в таблицу в виде ссылок на сохраненные файлы в static. Здесь используются шаблоны if и for, которые воспринимаются как условный оператор и цикл для того, чтобы в ячейках таблицы отобразить несколько выбранных изображений. В конце расположения есть кнопка submit, после нажатия которой осуществляется post-запрос на сервер.

Создадим файл forms.html. В данном html используется форма с элементами слайдера, которые позволяют выбирать значение из диапазона с помощью бегунка, а также элементы ввода строки и числового значения.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
 <h2>Форма ввода данных для изображений</h2>
    <form action="image_form" enctype="multipart/form-data" method="post">
        <p>
            Имя операции <br>
            <input name="name_op" value="operation" />
        </p>
        <p>
            Введите число <br>
            <input name="number_op" type="number" value = "0"  />
        </p>
        <table>
```

```
        <tr>
          <td><input type="range" id="r" name="r" value="123" oninput="x1.value=r.value" min="0" max="255" > </td>
          <td><output name="x1" for="r"> 123 </output></td>
        </tr>
        <tr>
          <td><input type="range" id="g" name="g" value="123" oninput="x2.value=g.value" min="0" max="255" > </td>
          <td><output name="x2" for="g"> 123 </output></td>
        </tr>
        <tr>
          <td><input type="range" id="b" name="b" value="123" oninput="x3.value=b.value" min="0" max="255" > </td>
          <td><output name="x3" for="b"> 123 </output></td>
        </tr>
        </table>
        <input name="files" type="file" multiple>
        <p>
        <input type="submit" value="Send" />
        </p>
      </form>
    {% if ready %}
    <table>
    {% for image in images %}
    <tr>
      <td>
        <img src = "{{ image }}">
      </td>
    </tr>
```

```
{% endfor %}
</table>
{% endif %}
</body>
</html>
```

Не забудьте добавить в Pipfile

```
python-multipart = "*"
```

или установить

```
pip install python-multipart
```

8 Примеры API функций

Для реализации REST API функций можно использовать pydantic. Здесь автоматически проводится проверка. Можно делать по аналогии с flasger и т. д.

```
from pydantic import BaseModel
class User(BaseModel):
    name: str
    age: int
@app.get('/users/{user_id}')
def get_user(user_id):
    return User(name="John Doe", age=20)
@app.put('/users/{user_id}')
def update_user(user_id, user: User):
    # поместите сюда код для обновления данных
    return user
```

ПОЛЕЗНЫЕ РЕСУРСЫ

1. https://www.perplexity.ai/
2. https://you.com/search?q=who+are+you&tbm=youchat&cfr=chat
3. https://docs.github.com/ru
4. https://fastapi.tiangolo.com/ru/