

Дипломная работа

Тема: Сравнение различных подходов к реализации REST API: Django Rest Framework, FastAPI и Flask-RESTful: Разработать REST API с использованием Django Rest Framework, FastAPI и Flask-RESTful, провести их сравнение.

Автор: Касымов Дмитрий Владимирович

Оглавление дипломной работы:

1. Введение
2. Основные понятия и определения
3. Методы и подходы к разработке
4. Архитектура приложения
5. Проектирование приложения
6. Разработка в соответствии с созданной документацией
7. Сравнение фреймворков Django Rest Framework, FastAP, Flask-RESTful

1.Введение

Обоснование выбора темы:

1. В современном мире любой сервис, который хочет масштабироваться и захватить “новую” целевую аудиторию, например в виде разработчиков ПО и крупных компаний, которые не готовы мириться с базовым функционалом клиентской части приложения, а стремятся автоматизировать работу интегрируя в свою экосистему все больше и больше возможностей требуют разработки своего REST API.
2. Потребности рынка: веб-разработчики, а также разработчики REST API пользуются растущим спросом т.к. почти любая крупная компания разрабатывает API своего сервиса, будь то открытое или закрытое, потому что: нет в интернете - нет в бизнесе.
3. Практическая значимость: Работа над API позволит серьезно прокачать навыки веб-разработки “с другой стороны” не загружая себя изучением дополнительных языков по типу: HTML, CSS, JS и их специфических фреймворков.
4. Личный интерес и потенциальные перспективы: лично мне очень интересно поработать по обе стороны разработки веб-приложений. Кроме того, такие навыки востребованы на рынке труда, что открывает перспективы для карьерного роста и развития.

Таким образом, выбор проекта REST API обусловлен ее актуальностью, потребностями рынка, практической значимостью и личным интересом, что делает эту тему подходящей для проведения дипломной работы

Определение цели и задач исследования:

Цель исследования: написать REST API для связи пользователь-поставленные задачи, а также сравнить фреймворки для его написания

Задачи исследования:

1. Обзор фреймворков для разработки REST API
2. Разработка сценариев сравнения: определить критерии сравнения и сформировать набор данных для первичной проверки.
3. Создание платформы и проведение тестирования: реализовать REST API сервиса для пары пользователь-поставленные задачи в разных вариациях технологий, и провести их тестирование.
4. Написание дипломной работы: составить дипломную работу, включающую в себя введение, обзор литературы, методологию и результаты исследования, анализ результатов, выводы и рекомендации.

Цели и задачи исследования направлены на получение практических результатов, которые позволят сформировать рейтинг технологий для написания REST API.

2. Основные понятия и определения

Обзор основных понятий в сервисе REST API для связи пользователь-задачи:

1. Фреймворк (Framework): Программная платформа, которая предоставляет готовые компоненты и инструменты для разработки приложений. В контексте REST API часто используются Django Rest Framework, FastAPI и Flask-RESTful.
2. REST (Representational State Transfer) API — это архитектурный стиль для построения веб-сервисов, основанный на использовании HTTP-протокола для обмена данными между клиентом и сервером.
3. Веб-приложение (Web Application): Программное приложение, которое работает на веб-сервере и доступно через браузер.
4. СУБД (Система Управления Базами Данных) — это программное обеспечение, предназначенное для создания, управления и хранения баз данных.
5. Данные — это любая информация, представленная в формализованном виде, пригодная для хранения, передачи, обработки и интерпретации человеком или автоматизированной системой. В контексте нашего проекта данные представляют собой информацию о товарах маркетплейса

3. Методы и подходы к разработке

Архитектура веб-приложения

1. Django REST Framework:
 - Использование моделей Django для представления сущностей (User, Task.)
 - Реализация ViewSets и Serializers для обработки HTTP-запросов и представления данных
 - Настройка маршрутизации с помощью DefaultRouter
 - Использование Django ORM для работы с базой данных

2. FastAPI:

- Определение Pydantic-моделей для представления сущностей
- Реализация CRUD-операций с использованием FastAPI-маршрутов
- Интеграция с СУБД с использованием SQLAlchemy
- Применение Pydantic для построения и валидации запросов/ответов

3. Flask-RESTful:

- Определение ресурсов (Resource) для обработки HTTP-методов
- Использование Flask-RESTful для маршрутизации и сериализации
- Интеграция с СУБД с применением SQLAlchemy
- Применение Marshmallow для сериализации и валидации данных

4. Архитектура приложения:

1. Модели:

- User: firstname, lasrname, email, job, age
- Task: title, content, priority, completed, user_id, slug

2. Эндпоинты:

- POST /user/: регистрация нового пользователя
- POST /task/: создание новой задачи
- DELETE /user/{user_id}/:удаление пользователя по id
- DELETE /task/{task_id}/:удаление задачи по id
- PATCH /products/{task_id}/: частичное изменение данных задачи
- GET/user/: получение списка пользователей
- GET/task/: получение списка задач
- GET/task/{task_id}: получение задачи по id
- GET/user/{user_id}: получение пользователя по id

3. Базовая структура:

- Уровень моделей (модели Django, Pydantic или SQLAlchemy)
- Уровень сервисов (бизнес-логика)
- Уровень контроллеров/представлений (обработка HTTP-запросов)
- Уровень маршрутизации (определение URL-адресов и HTTP-методов)
- Уровень взаимодействия с базой данных (Django ORM, SQLAlchemy)

Реализация данного приложения в трех вариантах (Django REST Framework, FastAPI и Flask-RESTful) позволит нам провести сравнительный анализ и выявить сильные и слабые стороны каждого подхода.

Особенности:

1. Django REST Framework:

- Django REST Framework (DRF) — это мощный и гибкий набор инструментов для построения веб-API на основе Django.
- Основные преимущества DRF:
 - Предоставляет готовые решения для CRUD-операций, сериализации, аутентификации и авторизации.
 - Поддержка различных форматов данных (JSON, XML, etc.).

- Встроенная документация API (Browsable API и Swagger).
- Расширяемость за счет большого сообщества разработчиков.
- Интеграция с Django ORM для работы с базой данных.
- Недостатки:
 - Относительно высокая сложность настройки и конфигурирования.
 - Может быть избыточным для простых API-приложений.
- 2. FastAPI:
 - FastAPI - современный, быстрый (с высокой производительностью) веб-фреймворк для построения API с использованием Python 3.6+.
 - Основные преимущества FastAPI:
 - Высокая производительность, основанная на ASGI-серверах (Starlette и Uvicorn).
 - Простота и быстрота разработки благодаря использованию типизированных Pydantic-моделей.
 - Автоматическая генерация документации (OpenAPI/Swagger).
 - Встроенная поддержка асинхронности.
 - Гибкая система аутентификации и авторизации.
 - Недостатки:
 - Относительно молодой фреймворк, меньшее сообщество и экосистема.
 - Может потребоваться больше настройки для интеграции с реляционными базами данных.
- 3. Flask-RESTful:
 - Flask-RESTful — это расширение к популярному Python-фреймворку Flask, предназначенное для построения RESTful API.
 - Основные преимущества Flask-RESTful:
 - Простота и минималистичность, основанные на базовом Flask.
 - Гибкость и возможность кастомизации под конкретные задачи.
 - Легковесность, подходит для небольших и средних API-приложений.
 - Хорошая интеграция с сторонними библиотеками, такими как Flask-JWT-Extended для аутентификации.
 - Недостатки:
 - Меньше готовых решений, чем в DRF, приходится реализовывать больше вручную.
 - Меньшая производительность по сравнению с FastAPI.
 - Может потребоваться больше времени на настройку аутентификации и авторизации.

5. Проектирование приложения

Планирование и анализ требований

- Выбор фреймворка и инструментов: Django Rest Framework, FastAPI, Flask-RESTful
- Определение структуры приложения: Разработка схемы архитектуры, включающей бэкенд, базу данных и интеграцию с внешними сервисами при необходимости.
- Разработка прототипа

- Реализация основного функционала: Создание базовой версии приложения с минимально необходимым функционалом для тестирования и демонстрации.
- Разработка ключевого функционала

Основные требования

Реализация REST API со следующим функционалом:

- Загрузка данных пользователей и задач в СУБД
- Получение данных о пользователях и задачах по id
- Изменение информации о пользователях и задачах
- Удаление задач, пользователей по id

Технические требования

- Бэкенд: Python с использованием фреймворков Django Rest Framework, FastAPI, Flask-RESTful
- База данных: Использование базы данных SQLite для хранения данных о пользователях/задачах и истории загрузок.

6. Разработка в соответствии с созданной документацией

Планирование разработки

Разработка была разделена на несколько основных этапов:

- Проектирование моделей баз данных
- Реализация серверной логики
- Разрабока функционала проверки и сериализации данных запросов
- Тестирование CRUD запросов разработанных приложений .
- Тестирование производительности приложений API, разработанных на FastAPI и Flask-REST-ful с помощью сторонней библиотеки Benchmark.js.

Технологии проекта:

Python 3.12

Django Rest Framework

FastAPI

Flask-RESTful

Разработка

Проектирование моделей баз данных

Созданы модели User и Task, связанные отношением между таблицами один ко многим и функционально с помощью relationship.

Пример созданных баз данных на Flask, с использованием классов, наследованных от базового класса SQLAlchemy.Model:

```
3  class User(db.Model):
4
5      __tablename__ = 'users'
6      id = db.Column(db.Integer(), primary_key=True)
7
8      firstname = db.Column(db.String(40), index=True)
9      lastname = db.Column(db.String(40), index=True)
10     email = db.Column(db.String(255), nullable=False, unique=True)
11     age = db.Column(db.Integer(), nullable=False)
12     job = db.Column(db.String(100), nullable=True)
13     task = db.relationship('Task', back_populates='user')
14
15     new *
16     def __repr__(self):
17         return f'<User {self.username}>'
18
19 7 usages new *
20  class Task(db.Model):
21
22     __tablename__ = 'tasks'
23     id = db.Column(db.Integer(), primary_key=True)
24
25     title = db.Column(db.String(128))
26     content = db.Column(db.String(250))
27     priority = db.Column(db.Integer())
28     completed = db.Column(db.Boolean())
29     user_id = db.Column(db.Integer(), db.ForeignKey('users.id'))
30     slug = db.Column(db.String(128))
31     user = db.relationship('User', back_populates='task')
32
33     new *
34     def save(self, *args, **kwargs):
35         ''' получение slug из title'''
36         if not self.id:
37             self.slug = slugify(self.title)
38             super(Task, self).save(*args, **kwargs)
39
40     new *
41     def __repr__(self):
42         return f'<Task {self.title}>'
```

Реализация серверной логики

Создан функционал обработки CRUD запросов на сервер.

Ниже представлен пример функционала, разработанного на фреймворке DRF с реализацией классического представления обработки запросов

```
58 class TaskViewSet(viewsets.ModelViewSet):
59     '''класс задач'''
60     new *
61     def get(self, request):
62         '''получение всех задач'''
63         tasks = Task.objects.all()
64         serializer = TaskSerializer(tasks, many=True)
65         return Response(serializer.data)
66     new *
67     def retrieve(self, request, task_id):
68         '''получение задачи по id'''
69         try:
70             task = Task.objects.get(id=task_id)
71         except Task.DoesNotExist:
72             return Response(data={"error": "User not found"}, status=404)
73         serializer = TaskSerializer(task)
74         return Response(serializer.data)
75     new *
76     def post(self, request):
77         '''post задачу'''
78         serializer = TaskSerializer(data=request.data)
79         if serializer.is_valid():
80             serializer.save()
81             return Response(serializer.data, status=201)
82         return Response(serializer.errors, status=400)
83     new *
84     def put(self, request, task_id):
85         '''корректировка задачи'''
86         try:
87             user = Task.objects.get(id=task_id)
88         except Task.DoesNotExist:
89             return Response(data={"error": "Task not found"}, status=404)
90         serializer = TaskSerializer(data=request.data)
91         if serializer.is_valid():
92             task = serializer.data
93             return Response(task, status=201)
94         return Response(serializer.errors, status=400)
```

Разработка функционала проверки и сериализации данных CRUD запросов

На фреймворках DRF, FastAPI и Flask Rest-ful разработан функционал проверки и сериализации данных CRUD запросов.

Ниже представлен пример валидации данных CRUD запросов, выполненный на базе классов, наследованных от `pydantic.BaseModel`. Программный код написан на FastAPI.

```
2 usages new *
4 class CreateUser (BaseModel):
5
6     firstname: str = Field(min_length=2)
7     lastname: str = Field(min_length=2)
8     email: EmailStr
9     age: int = Field(gt=16,lt=99)
10    job: str = Field(min_length=2)
11
12
13
14 2 usages new *
15 class UpdateUser (BaseModel):
16     firstname: str = Field(min_length=2)
17     lastname: str = Field(min_length=2)
18     email: EmailStr
19     age: int = Field(gt=16,lt=99)
20     job: str = Field(min_length=2)
21
22
23 2 usages new *
24 class CreateTask (BaseModel):
25     title: str
26     content: str
27     priority: int = Field(gt=0)
28     completed: bool = Field(default=False)
29
30
31 2 usages new *
32 class UpdateTask (BaseModel):
33     title: str
34     content: str
35     priority: int = Field(gt=0)
36     completed: bool = Field(default=False)
37     user_id: int = Field(gt=0)
38
39
40 new *
41 class Config:
42     orm_mode = True
```


Тестирование CRUD запросов разработанных приложений .

Тестирование разработанных приложений выполнено с помощью библиотеки requests. Произведены CRUD запросы на локальный сервер компьютера, результаты представлены в виде кода ответа, времени исполнения и текста запрашиваемых данных.

Ниже приведен пример реализации GET запроса получения списка пользователей

```
1 import requests
2 import time
3
4 url = "http://127.0.0.1:5000/"
5 headers = {'Authorization': 'Bearer example-auth-code'}
6
7 start_time = time.time()
8 response = requests.get(url+'users', headers=headers)
9 duration = time.time() - start_time
10
11 print(f"Время исполнения - {duration:.3f} sec")
12 print(response, response.text)
```

Run main x users_all x

C:\Users\kolya\PycharmProjects\Flask3\ven\Scripts\python.exe C:\Users\kolya\PycharmProjects\Flask3\tests\users_all.py
Время исполнения - 1.053 sec
<Response [200]> [
 {
 "id": 3,
 "firstname": "kola",
 "lastname": "Ponomarev",
 "email": "kokakola@example.com",
 "age": 25,
 "job": "worker"
 },
 {
 "id": 4,
 "firstname": "Any",
 "lastname": "Ponomarev",
 "email": "anya@example.com",
 "age": 23,
 "job": "worker"
 },
 {
 "id": 5,
 "firstname": "bom",
 "lastname": "kas",
 "email": "bom@example.com",
 "age": 24,
 "job": "programmer"
 }
]

x3 > tests > users_all.py

Тестирование производительности приложений API, разработанных на FastAPI и Flask-REST-ful с помощью сторонней библиотеки Benchmark.js.

В данном функционале выполнено тестирование производительности REST API. В обоих приложениях с использованием библиотеки Benchmark.js. выполнены POST запросы на локальный сервер – создание нового пользователя:

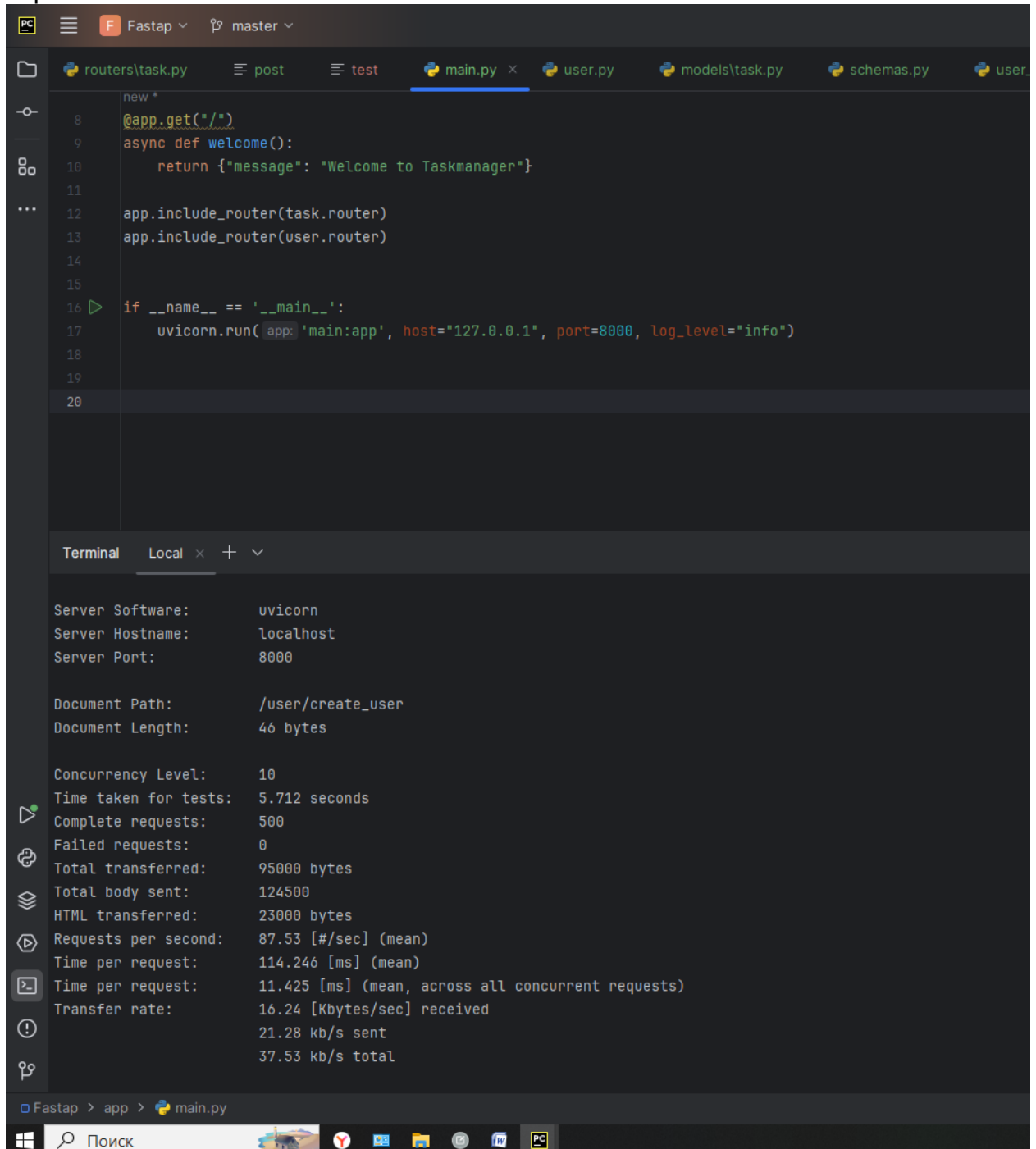
```
{"firstname": "bom", "lastname": "kas", "email": "bom@example.com", "age": 24, "job": "programmer"}
```

POST состоял из серии 500 запросов партиями по 10 параллельных запросов. Данный тест проведен мною 3 раза. По результатам тестов асинхронный FastAPI, как и ожидалось показал меньшее время.

Среднее время теста Fast API – 5,63 сек

Среднее время теста Flask – 10,52 сек

Скриншот теста на FastAPI:



The screenshot shows a code editor with a Python file named `main.py` in the `Fastap` project. The code defines a FastAPI application with a single endpoint `@app.get("/")` that returns a JSON message. The application is run using `uvicorn` on `localhost:8000`.

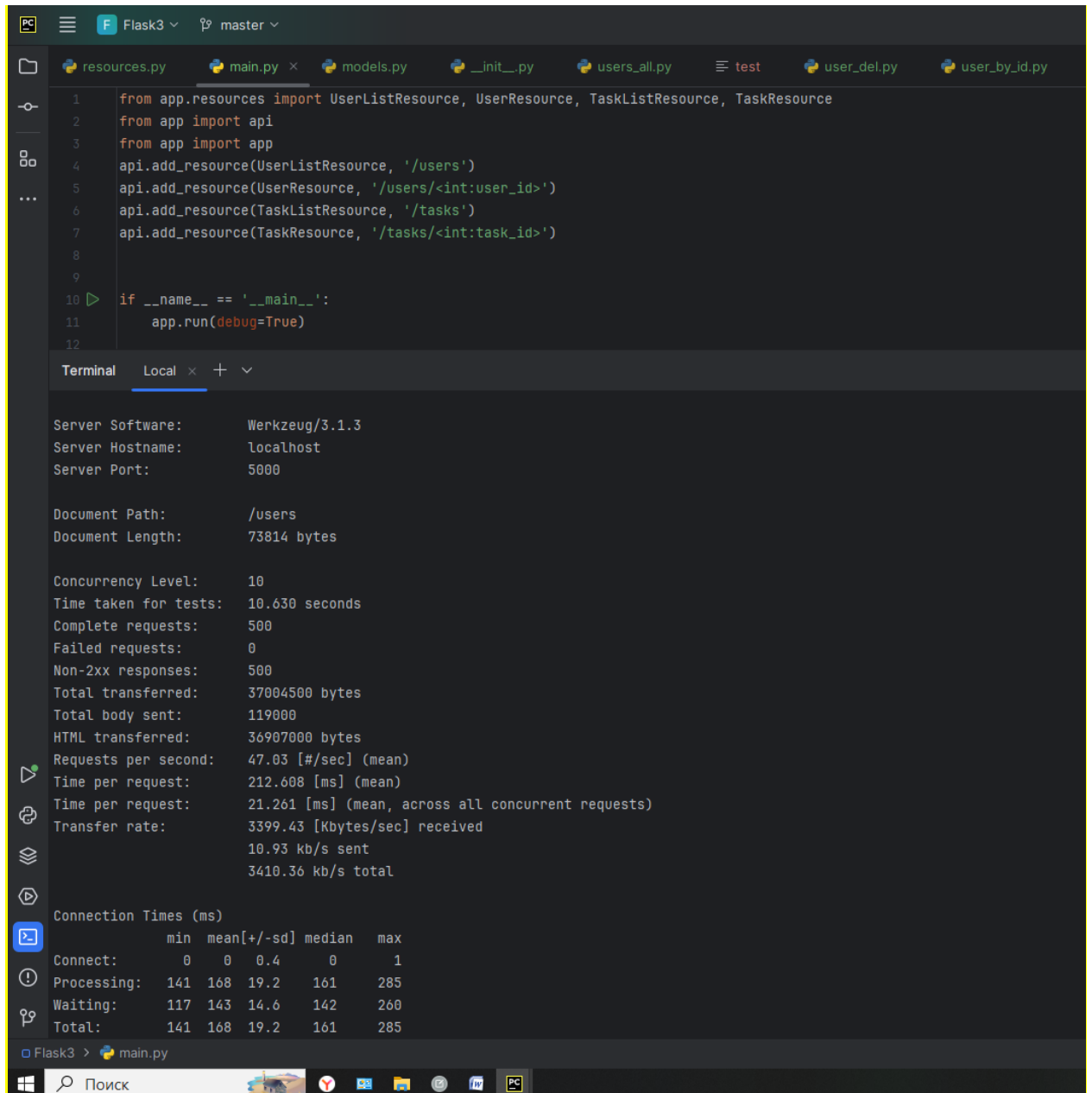
Below the code editor, a terminal window displays the output of a test run. The test results show that the application successfully handled 500 requests in 5.712 seconds, with a mean time per request of 11.425 ms.

```
Server Software:      uvicorn
Server Hostname:      localhost
Server Port:          8000

Document Path:        /user/create_user
Document Length:       46 bytes

Concurrency Level:     10
Time taken for tests:  5.712 seconds
Complete requests:     500
Failed requests:        0
Total transferred:     95000 bytes
Total body sent:       124500
HTML transferred:      23000 bytes
Requests per second:   87.53 [#/sec] (mean)
Time per request:      114.246 [ms] (mean)
Time per request:      11.425 [ms] (mean, across all concurrent requests)
Transfer rate:         16.24 [Kbytes/sec] received
                       21.28 kb/s sent
                       37.53 kb/s total
```

Скриншот теста на Flask:



The screenshot shows a VS Code editor with a Flask application open. The file explorer on the left shows a project named 'Flask3' with a 'master' branch. The editor displays the following Python code in 'main.py':

```
1 from app.resources import UserListResource, UserResource, TaskListResource, TaskResource
2 from app import api
3 from app import app
4 api.add_resource(UserListResource, '/users')
5 api.add_resource(UserResource, '/users/<int:user_id>')
6 api.add_resource(TaskListResource, '/tasks')
7 api.add_resource(TaskResource, '/tasks/<int:task_id>')
8
9
10 if __name__ == '__main__':
11     app.run(debug=True)
12
```

The terminal window at the bottom shows the output of the application, including server information and performance metrics:

```
Server Software: Werkzeug/3.1.3
Server Hostname: localhost
Server Port: 5000

Document Path: /users
Document Length: 73814 bytes

Concurrency Level: 10
Time taken for tests: 10.630 seconds
Complete requests: 500
Failed requests: 0
Non-2xx responses: 500
Total transferred: 37004500 bytes
Total body sent: 119000
HTML transferred: 36907000 bytes
Requests per second: 47.03 [#/sec] (mean)
Time per request: 212.608 [ms] (mean)
Time per request: 21.261 [ms] (mean, across all concurrent requests)
Transfer rate: 3399.43 [Kbytes/sec] received
10.93 kb/s sent
3410.36 kb/s total

Connection Times (ms)
min mean[+/-sd] median max
Connect: 0 0 0.4 0 1
Processing: 141 168 19.2 161 285
Waiting: 117 143 14.6 142 260
Total: 141 168 19.2 161 285
```

The bottom status bar shows the current file is 'main.py' in the 'Flask3' project.

7. Сравнение фреймворков Django Rest Framework, FastAP, Flask-RESTful

Критерии сравнения:

1. Поддержка асинхронности - способность обрабатывать асинхронные запросы
2. Уровень безопасности - возможности для реализации безопасности приложения
3. Сложность настройки - уровень сложности развертывания и настройки фреймворка

4. Скорость разработки - скорость и удобство создания API с использованием фреймворка
5. Интеграция с базами данных - легкость работы с различными типами баз данных
6. Скорость обработки запросов

Критерий	DRF	Fastapi	Flask-RESTful
Поддержка асинхронности	<ul style="list-style-type: none"> • DRF не поддерживает асинхронную обработку запросов 	<ul style="list-style-type: none"> • FastAPI имеет встроенную поддержку асинхронности и позволяет создавать асинхронные API 	<p>Фреймворк Flask - RESTful, как приложение WSGI, использует одну задачу/воркер для обработки одного цикла запроса/ответа</p>
Уровень безопасности	<ul style="list-style-type: none"> • DRF предлагает множество встроенных инструментов для обеспечения безопасности приложения 	<ul style="list-style-type: none"> • FastAPI не имеет встроенной системы безопасности в традиционном понимании. Вместо этого он предоставляет модуль <code>fastapi.security</code>, который включает в себя ряд классов для аутентификации и работы с API ключами 	<ul style="list-style-type: none"> • Flask предоставляет разработчикам основу для реализации множества функций безопасности, но многие аспекты безопасности требуют ручной настройки или интеграции с внешними библиотеками.
Сложность настройки	<ul style="list-style-type: none"> • DRF может быть сложен в настройке и требует определенных знаний Django 	<ul style="list-style-type: none"> • FastAPI, благодаря своей простоте и удобству использования, позволяет быстро создавать API 	<p>Flask — это легковесный веб-фреймворк для Python, который широко используется для создания веб-приложений и API.</p>
Скорость разработки	<ul style="list-style-type: none"> • DRF имеет больше инструментов и функциональностей, что может замедлить скорость разработки 	<ul style="list-style-type: none"> • FastAPI, благодаря своей простоте и удобству использования, позволяет быстро создавать API 	<p>Flask -RESTful, благодаря своей простоте и удобству использования, позволяет быстро создавать API</p>
Интеграция с базами данных	<ul style="list-style-type: none"> • DRF легко интегрируется с базами данных с 	<ul style="list-style-type: none"> • FastAPI также обладает хорошей интеграцией с 	<p>Flask поддерживает интеграцию с</p>

	использованием Django ORM	базами данных и предоставляет возможность выбора ORM	множеством различных баз данных, включая SQL и NoSQL. Выбор базы данных зависит от нужд вашего приложения.
Скорость обработки запросов		За счет поддержки асинхронности FastAPI выигрывает в производительности обработки запросов у своих конкурентов.	

Исходя из этого сравнения, можно заключить, что FastAPI предлагает более высокую производительность, легкость работы с асинхронностью и безопасностью, а также удобство настройки и создания API. Flask –RESTful, также выделяется легкостью разработки, организованностью кода, а также удобством настройки и создания API. В то время как DRF может быть более удобным для разработчиков, знакомых с Django, и предоставляет больше инструментов и функциональностей для расширения приложения. Выбор между ними будет зависеть от специфики проекта и требований к разрабатываемому API.