Итоговый проект

Тема: «Сервис микроблогов».

Легенда

Вы — главный бэкенд-разработчик на Python известной веб-студии и уже привыкли к тому, что полностью отвечаете за бэкенд. Архитектура приложения, реализация бизнес-логики, тестирование и развёртывание — всё то, что вы любите и умеете отлично делать. Основная специализация веб-студии — создание лендингов, айдентики, CMS, поэтому вы легко можете в одиночку вести несколько проектов, так как основная нагрузка ложится на фронтенд. В итоге ваша работа выглядит так: получить описание задачи от CEO, договориться или получить уже готовый контракт API от фронтенд-разработчика и реализовать бэкенд под этот фронт.

И вот на волне импортозамещения один местный бизнесмен, который наслышан о качестве работы вашей студии, решил прийти к вам с заказом на реализацию собственного корпоративного сервиса микроблогов, очень похожего на Twitter. Таких заказов сейчас неимоверно много, поэтому вы оказались загружены, а согласование требований и контракта прошло заочно: вам предоставили техническое задание. Поскольку вы были заняты на другом проекте, а фронтенд-разработчик был свободен, вы решили поручить ему разработку контракта, что он успешно сделал, успев написать ещё и фронтенд.

В итоге, когда вы приступили к этому проекту, в вашем распоряжении были готовый фронтенд, описание API и горящие сроки (как же без этого). Теперь ваша очередь показать класс и написать бэкенд под этот непростой проект. Готовы?

Техническое задание

Цель задачи

Реализовать бэкенд сервиса микроблогов.

Описание поведения

Для корпоративного сервиса микроблогов необходимо реализовать бэкенд приложения. Поскольку это корпоративная сеть, то функционал будет урезан относительно оригинала. Как правило, описание сервиса лучше всего дать через функциональные требования, то есть заказчик формулирует простым

языком, **что** система должна уметь делать. Или что пользователь хочет делать с системой. И вот что должен уметь делать наш сервис:

Функциональные требования:

1. Пользователь может добавить новый твит.
2. Пользователь может удалить свой твит.
3. Пользователь может зафоловить другого пользователя.
4. Пользователь может отписаться от другого пользователя.
5. Пользователь может отмечать твит как понравившийся.
6. Пользователь может убрать отметку «Нравится».
7. Пользователь может получить ленту из самых популярных твитов от пользователей, которых он фоловит.
8. Твит может содержать картинку.

Заметим, что требования регистрации пользователя нет: это корпоративная сеть и пользователи будут создаваться не нами. Но нам нужно уметь отличать одного пользователя от другого. Об этом поговорим чуть позже.

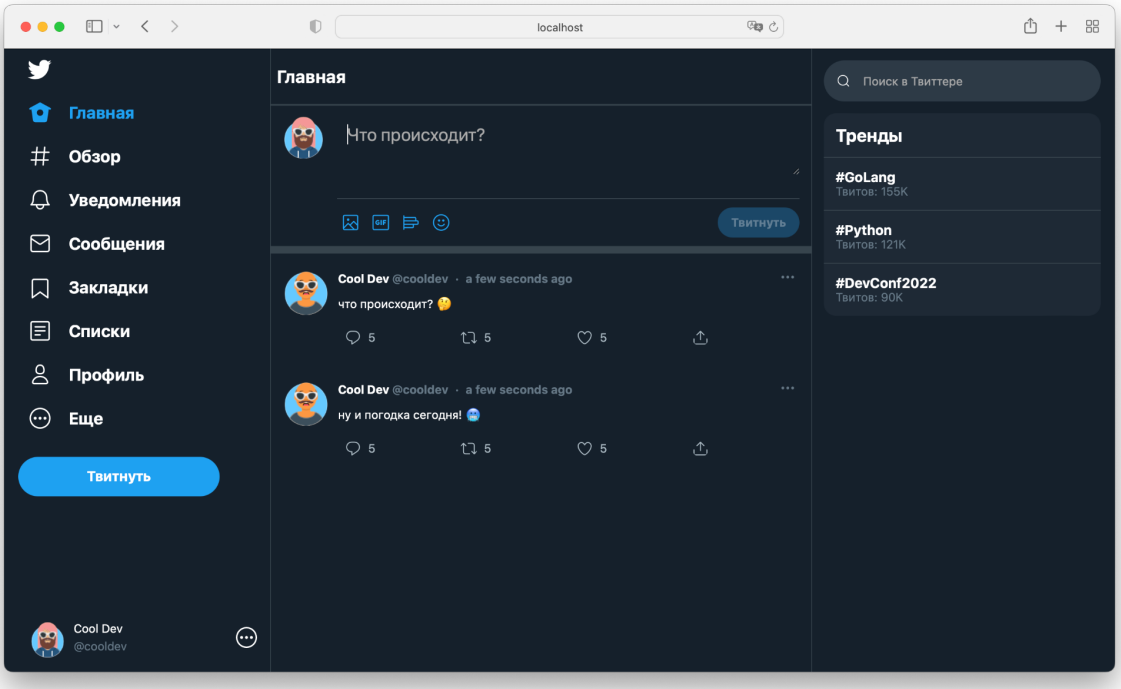
Также систему описывают через список нефункциональных требований, то есть список того, **как** система должна выполнять функциональные требования.

Нефункциональные требования:

1. Систему должно быть просто развернуть через Docker Compose.
2. Система не должна терять данные пользователя между запусками.
3. Все ответы сервиса должны быть задокументированы через Swagger. Документация должна быть доступна в момент запуска приложения. Также не забудьте оформить развёрнутый README с описанием проекта и инструкцией по запуску приложения.

Состав продукта:

Основным потребителем нашего контента будет вот такая страничка:



Для этого нужно будет написать несколько endpoint.

Первый и самый важный:

1.

POST /api/tweets

HTTP-Params:

api-key: str

{

“tweet_data”: string

“tweet_media_ids”: Array[int] // Опциональный параметр. Загрузка картинок будет происходить по endpoint /api/media. Фронтенд будет подгружать картинки туда автоматически при отправке твита и подставлять id оттуда в json.

}

Запросом на этот endpoint пользователь будет создавать новый твит. Бэкенд будет его валидировать и сохранять в базу.

В ответ должен вернуться id созданного твита.

{

“result”: true,

“tweet_id”: int

}

2.

Endpoint для загрузки файлов из твита. Загрузка происходит через отправку формы.

POST /api/medias

HTTP-Params:

api-key: str

form: file=”image.jpg”

В ответ должен вернуться id загруженного файла.

{
“result”: true,

“media_id”: int

}

3.

Ещё нам потребуется endpoint по удалению твита. В этом endpoint мы должны убедиться, что пользователь удаляет именно свой собственный твит.

DELETE /api/tweets/<id>

HTTP-Params:

api-key: str

В ответ должно вернуться сообщение о статусе операции.

{

“result”: true

}

Пользователь может поставить отметку «Нравится» на твит.

POST /api/tweets/<id>/likes

HTTP-Params:

api-key: str

В ответ должно вернуться сообщение о статусе операции.

{

“result”: true

}

4.

Пользователь может убрать отметку «Нравится» с твита.

DELETE /api/tweets/<id>/likes

HTTP-Params:

api-key: str

В ответ должно вернуться сообщение о статусе операции.

{

“result”: true

}

5.

Пользователь может зафоловить другого пользователя.

POST /api/users/<id>/follow

HTTP-Params:

api-key: str

В ответ должно вернуться сообщение о статусе операции.

{

“result”: true

}

6.

Пользователь может убрать подписку на другого пользователя.

DELETE /api/users/<id>/follow

HTTP-Params:

api-key: str

В ответ должно вернуться сообщение о статусе операции.

{

“result”: true

}

7.

Пользователь может получить ленту с твитами.

GET /api/tweets

HTTP-Params:

api-key: str

В ответ должен вернуться json со списком твитов для ленты этого пользователя.

```
{
  “result”: true,
  "tweets": [
    {
      "id": int,
      "content": string,
      "attachments" [
         link_1, // relative?
         link_2,
         ...
       ]
      "author": {
         "id": int
         "name": string
       }
      “likes”: [
```

```
        {
         “user_id”: int,
         “name”: string
        }
      ]
   },
   ...,
  ]
}
```

В случае любой ошибки на стороне бэкенда возвращайте сообщение следующего формата:

{

“result”: false,

“error_type”: str,

“error_message”: str

}

8.

Пользователь может получить информацию о своём профиле:

GET /api/users/me

HTTP-Params:

api-key: str

В ответ получаем:

```
{
   "result":"true",
   "user":{
      "id":"int",
```

```
      "name":"str",
      "followers":[
         {
            "id":"int",
            "name":"str"
         }
      ],
      "following":[
         {
            "id":"int",
            "name":"str"
         }
      ]
   }
}
```

9.

Пользователь может получить информацию о произвольном профиле по его id:

GET /api/users/<id>

В ответ получаем:

```
{
   "result":"true",
   "user":{
      "id":"int",
      "name":"str",
```

```
        "followers":[
            {
                "id":"int",
                "name":"str"
            }
        ],
        "following":[
            {
                "id":"int",
                "name":"str"
            }
        ]
    }
}
```

Технические требования

База данных: используйте базу данных PostgreSQL. Храните в ней информацию о твитах, пользователях, лайках, фоловерах и медиа.

Документация: как вы заметили, в ТЗ документация не совсем в обычном виде. Всё дело в том, что фронтенд-разработчик — новичок и не знает, как принято оформлять документы в вашей компании. Давайте это исправим. Все ответы сервиса должны быть задокументированы через Swagger. Документация должна быть доступна в момент запуска приложения. Также не забудьте оформить развёрнутый README с описанием проекта и инструкцией по запуску приложения.

Деплой: запуск приложения должен происходить по команде docker-compose up -d.

Тестирование: приложение должно быть покрыто unit-тестами, проверено линтером.

Полуопционально: mypy. Написать код так, чтобы он удовлетворил всем требованиям mypy, сложно. Но постарайтесь, чтобы статический типизатор не слишком сильно ругался на ваш код. В идеале чтобы ошибок не было вообще.

Формат сдачи материалов и оценивание

Результат проведённой работы опубликуйте на GitLab и отправьте ссылку через форму ниже. Убедитесь, что предоставили доступ к репозиторию и проект полностью готов к тому, чтобы быть развёрнутым.

Критерии оценивания

- Проект полностью соответствует ТЗ. Все функциональные и нефункциональные требования имплементированы и протестированы. Приложение работает без ошибок (нет неотловленных exception).
- Проект легко развернуть в одну-две команды на любой машине.
- Pythonic-код легко читаем, идиоматичен, проверен линтерами и соответствует их требованиям. Есть вся необходимая документация функций, методов, классов и модулей, присутствует аннотация типов.
- Есть файл README.md, в котором для пользователя описывается подробная инструкция по эксплуатации сервиса.

Подсказки и советы

- Можете проверить качество тестов через метрику покрытия: сколько процентов кода покрыто тестами? Используйте для этого утилиту.
- Чтобы сделать красивый README.md, можете воспользоваться гайдлайном.
- Не забудьте разделить dev- и prod-окружение. Это значит, что в запущенном приложении, скорее всего, не понадобятся библиотеки линтеров и тестов.
- В документации Flask вы найдёте информацию о том, как делать загрузку файлов во flask-приложении. Во всех остальных фреймворках похожий принцип работы.
- В шаблон диплома уже внесены минимальные настройки для запуска UI сайта. Используйте их и сделайте другие по своему усмотрению.