Пояснительная записка по проекту “Rest-api приложение на Flask”.

Идея проекта: создать веб-сервис для работ по всему миру.

Требования к идее: сервис должен позволять пользователю регистрироваться, входить в свой аккаунт, изменять данные аккаунта, добавлять работу

Ссылка на приложение в облаке: <https://yl-flask-alice.herokuapp.com/>

Папки и файлы проекта:

* alembic – папка управления миграций
* data – в ней находится файл создания движка sqlaclhemy и файл предоставляющий модели для добавления в БД при создании
* db – папка с БД
* models – папка с моделями пользователей, работ, сообщений и категорий работ
* rest-api – папка с ресурсами и файлом парсеров аргументов
* static: img – папка с аватарками пользователей. Загрузка с помощью сохранение с помощью file.save(path);

css – файлы со стилями css

* templates – папка с шаблонами. Главный шаблон – main\_template.html, остальные его дополняют
* alembic.ini – файл конфигурации alembic
* forms\_for\_page.py – файл с классами форм для шаблонов
* main\_app.py – файл с приложением и обработчиками страниц
* requirements.txt – файл с нужными библиотеками
* Rest-api приложение на Flask.pptx – презентация проекта
* tests.py – файл с тестирующими функциями

Модели:

* users
  + id – id пользователе
  + surname – фамилия пользователя
  + name – имя пользователя
  + age – возраст пользователя(число)
  + speciality – специализация
  + about – пользователь о себе
  + email – почта(уникальна)
  + hashed\_password - захешированный пароль(добавляется с помощью метода set\_password)
  + avatar\_url – url к аватарке пользователя
  + created\_date - время создания аккаунта
  + chats – чаты. Чаты разделяются “;”, а id получателя и id сообщения “, ”(“2, 1; 4, 6”)
  + job – список связанных Jobs элементов
  + message – список связанных Message элементов

функция set\_password – добавляет захешированный пароль к пользователю

функция check\_password – возвращает сравнение переданного пароля, с паролем пользователя

* jobs
  + id – id работы
  + author – id автора(Внешний ключ)
  + header – заголовок
  + requirements – требования
  + description – описание
  + creation\_date – время создания работы
  + user – список User обьектов
  + categories– список категорий работы
* messages
  + id – id работы
  + sender – id отправителя(Внешний ключ)
  + receiver – id получателя
  + text – текст сообщения
  + date – время обновления сообщения
  + user – список User объектов
* categories
  + id – id категории
  + name – наименование категории

создаётся промежуточная таблица association:

* jobs – id работы(Внешний ключ)
* category - id категории(Внешний ключ)

Функции и обработчики:

файл main\_app.py

* main(запуск приложения):
  + создаёт движок и создаёт новую БД(если нет таковой) в db/work\_db.sqlite.
  + добавляет ресурсы в api приложение
  + запуск приложения
* register(“/register”)
  + инициализация формы LoginForm
  + Если валидация прошла успешно, то делается POST запрос в UsersListResource с аргументами в виде данных формы.
  + Если пользователь добавился в бд, то идём на страницу авторизации. Иначе выводим ошибку
* login(“/login”)
  + инициализация формы LoginForm
  + Если валидация прошла успешно, то сначала находится пользователь в бд с данными почтой и паролем. Если какие-то данные введены неправильно то ошибка
  + Если все правильно – переходим на главную страницу сервиса
  + Отправляем шаблон
* main\_page(“/”)
  + Сначала берём очередь(query) работ из бд
  + Отправляем шаблон с работами и их авторами
* load\_user
  + возвращает текущего пользователя – current\_user
* logout(“/logout”)
  + выходит из аккаунта пользователя
* add\_job(“add\_job”)
  + инициализация формы JobForm
  + для использования множественных чекбоксов как одно целое я добавил класс MultiCheckboxField в JobForm.select и добавляю значения для чекбоксов [(value, label), (value1, label1) …]
  + при успешной валидации отправляю POST запрос в JobsListResource с аргументами: заголовок, требования, описание, id автора и словарь для категорий {выборка, данные заполнения, данные чекбокса новой работы и наименование новой работы}
  + возвращение на главную страницу
* edit\_job(“/edit\_job/<int:id>”)
  + проверка на существование работы и проверка на случай если не автор изменяет работу
  + инициализация формы JobForm
  + если идёт запрос то отправлям GET запрос в JobsResource с id работы
  + дальше как в add\_job, только вместо POST запроса отправляется PUT запрос в JobsResource с id работы
* delete\_job (“/delete\_job/<int:id>”)
  + проверка на существование работы и проверка на случай если не автор изменяет работу
  + отправка DELETE запроса в JobsResource с id работы
* edit\_profile(“/profile”)
  + инициализация формы MyProfile
  + если идёт запрос то отправлям GET запрос в UsersResource с id пользователя и заполняем форму
  + Если валидация прошла успешно:
    - проверка на правильный ввод старого пароля. Если неправильный пароль – шаблон с ошибкой
    - получаем картинку из данных запроса request.files["image"] по id элемента
    - если файл добавили, то сначала создаётся защищённая версия файла, к которой можно обратиться с разных ос, возвращается путь к файлу
    - файл сохраняется в папке app.config['UPLOAD\_FOLDER']
    - создаётся url к файлу(параметр – возвращаемое значение функции upload\_filename)
    - PUT запрос в UsersResource с id пользователя
    - возвращение на главную страницу
* messaging(“/open\_chat/<int:receiver\_id>”)
  + проверка на то, что мы отправляем сообщение не себе
  + инициализация формы ChatForm
  + если request.method == "GET", то получаем переписку с пользователем(id= receiver\_id) и получаем этого пользователя. Дальше всё отправляем в шаблон
  + если форма проходит валидацию то отправляем сообщение(POST запрос в MessagesListResource)
* edit\_message(“/edit\_message/<int:message\_id”)
  + инициализация формы ChatForm
  + меняю текст кнопки отправки сообщений(мы меняем, а не отправляем сообщение)
  + если request.method == "GET", то получаем сообщение по message\_id, заполняем поле отправки сообщения текстом, получаем переписку с пользователем(id= receiver\_id). Дальше всё отправляем в тот же шаблон что и у messaging(получателя получаем по аргументу ответа resp: session.query(User).get(resp["receiver"]))
  + если форма проходит валидацию то изменяем сообщение(PUT запрос в MessagesResource с message\_id)
  + возвращаемся к обработчику чата “/open\_chat/<int:receiver\_id>”. Параметр-полученный из GET запроса к ресурсу Сообщений(MessagesResource) аргумент "receiver"
* delete\_message (“/delete\_message/<int:message\_id> ”)
  + GET запрос к ресурсу Сообщений(MessagesResource)
  + удаляем сообщение(DELETE запрос) по message\_id
  + возвращаемся к обработчику чата “/open\_chat/<int:receiver\_id>”. Параметр-аргумент "receiver" полученный из GET запроса resp
* chats (“/chats ”) # показывает список чатов с последним сообщением в них
  + получаем список кортежей id получателя, id сообщения из current\_user.chats атрибута(заранее обернув в функцию возвращающую список разделённых чатов)
  + получаем список пользователей по id получателя из кортежей chats, с которыми имеется переписка
  + получаем список сообщений последних сообщений чатов по id сообщения из кортежей chats
  + отправляем шаблон
* view\_profile(“/profile/<int:user\_id>”)
  + Всё то же самое как в edit\_profile, только отправляем другой шаблон, который обрабатывает не все элементы формы. Параметр id для проверки кто смотрит профиль: мы или не мы

Rest-api обработчики(в конце каждой get, post, put, delete функции - session.commit())

файл users\_resources.py

* UsersResource:
  + get
    - проверка наличия пользователя
    - получение пользователя по id
    - возвращение атрибутов пользователя
  + put
    - парсинг аргументов
    - проверка наличия пользователя
    - получение пользователя по id
    - если пользователь меняет почту 🡪 проверка на уникальность почты
    - изменение атрибутов пользователя
    - если есть фото: изменяем avatar\_url
    - если есть пароль: изменяем password
* UsersListResource:
  + get
    - возвращается список словарей атрибутов
  + post
    - парсинг аргументов
    - проверка на уникальность почты
    - создание класса пользователя User(avatar\_url при создании ссылается на аватарку анонима)

файл jobs\_resources.py

* JobsResource:
  + get
    - проверка наличия работы
    - получение работы по id
    - возвращение атрибутов работы(атрибут categories в удобной форме для form.select.value: [(value, label), (value1, label1) …])
  + delete
    - получение работы по id
    - удаление работы
  + put
    - парсинг аргументов
    - проверка наличия работы
    - получение работы по id
    - изменение атрибутов работы
    - 2 списка новых и старых категорий
    - удаляем категории которые пропали
    - добавляем категории которые появились
    - если есть чекбокс новой категории, то создаём категорию, присваиваем ей имя и добавляем в список категорий
* JobsListResource:
  + get
    - возвращение списка словарей атрибутов работы(атрибут categories в удобной форме для form.select.value: [(value, label), (value1, label1) …])
  + post
    - парсинг аргументов
    - создание класса работы Jobs
    - из списка категорий добавляются выбранные
    - если есть чекбокс новой категории, то создаём категорию, присваиваем ей имя и добавляем в список категорий

файл messages\_resources.py

* MessagesResource:
  + get
    - проверка наличия сообщения
    - получение сообщения по id
    - возвращение атрибутов сообщения(получателя и текста сообщения)
  + delete
    - получение сообщения по id
    - получение класса отправителя и класса получателя

Здесь будет также изменяться атрибут User.chats для каждого из собеседников – поэтому берём два класса пользователей

* + - представление чатов в более удобном формате(список кортежей: получатель, сообщение)
    - получение списка сообщений
    - итерация по списку сообщений:
      * если сообщение с соответствующим id и оно последнее, то если это единственное сообщение – удаляем эти чаты у обоих пользователей, иначе изменяем значение id последнего сообщения на id предшествующего сообщения у обоих пользователей
      * перевожу списки чатов обратно в строковые форматы для пользователей
    - удаление сообщения
  + put
    - парсинг аргументов
    - если сообщение не пустое
      * получение сообщения по id
      * изменение атрибутов сообщения
* MessagesListResource:
  + get
    - парсинг аргументов
    - получение сообщений где (отправитель = мой id и получатель = id собеседника) или (отправитель = id собеседника и получатель = мой id)
    - переводив в список кортежей атрибутов(id, id отправителя, id получателя, текст, время в строковом формате)
  + post
    - парсинг аргументов
    - если сообщение не пустое
      * создаём класс сообщений Message
      * заполняем атрибуты
      * добавляю класс в сессию чтобы пользоваться атрибутом id
      * получаю отправителя и получателя
      * изменяю их чаты в список кортежей
      * если уже есть переписка с собеседником, то изменяю id последнего сообщения на id нового сообщения и добавляю чат в начало списка и делаю то же самое для получателя.
      * А если нет переписки, то создаю её в начале чатов пользователей
      * перевожу списки чатов обратно в строковые форматы для пользователей