

Замечания к второй лабораторной работе

Важные замечания:

- Понадобилось пересмотреть файловую структуру проекта для того, чтобы всё заработало: теперь в `app` находится всё кроме папки `alembic` и `main.py`, в самой же папке `app` находится `__init__.py`, из-за которого `python` при импорте воспринимает его как модуль и может добраться до `models.py` и достать оттуда конструктор таблицы `User`.
- В файле `main.py` (или любое другое имя которым назван файл, который запускает сервер FastAPI) можно добавить код ниже и тогда можно запускать из консольной командой `python main.py` сервер. Только для этого нужно создать в папке `app/api` `py`-файл где лежат обычные константные переменные хранящие информацию о порте и `ip` сервера (туда же можно добавить эндпоинты):

```
if __name__ == "__main__":  
    uvicorn.run(app, host=FastApiServerInfo.IP,  
port=FastApiServerInfo.PORT)
```

- Если появится желание (или нужда) добавить новую таблицу в базу данных которую смотрит `alembic`, то:
 - Редактируем файл где хранятся конструкторы структур таблиц, в моём случае этом `models.py`
 - Теперь в командной строке `alembic upgrade head` и `alembic revision --autogenerate -m "some test"`

Сервер - это `main.py`, запускать из терминала командой `uvicorn main:app --host 127.0.0.1 --port 12000`, где **server_script** - название `py` файла, **server_app** - название переменной которой присвоено FastAPI.

Alembic - по сути `git`, но для бд в рамках одного проекта

Для того чтобы добавить `alembic` в проект делаем следующее:

- Переходим через консоль в папку проекта и пишем `alembic init alembic`
- Создаётся папка `alembic`, в ней находим `alembic.ini`, ищем строку `sqlalchemy.url = ...` и пишем `sqlite:/// (относительный путь к бд, которой желательно лежать в папке проекта)`
Очень важно докинуть в папку `app` файл `init.py` (он может быть пустым), иначе `python` не будет понимать, что из `app` можно что-то экспортировать
- В файле `env.py` импортируем модель SQLAlchemy, например так `from app.models import (класс которым назван конструктор таблицы)` и редактируем строку `target_metadata = (класс которым назван конструктор таблицы).metadata`

Теперь заставляем это работать:

- В командной строке пишем `alembic revision --autogenerate -m "create users table"` - это создаст в `alembic/versions` py-файл. Его можно отредактировать (а именно методы **upgrade** и **downgrade**), если есть желание и умение. Это у нас создалась та самая миграция (или коммит).
- Всё в той же командной строке пишем `alembic upgrade head` для применения миграции (коммита) и `alembic downgrade -1` чтобы откатиться на предыдущую миграцию (коммит)

SQLAlchemy - автоматизация создания структур бд через python.

Пример кода ниже пишем в отдельном файле, а потом из него импортируем нужный класс.

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

# Создаём базовый класс для моделей
Base = declarative_base()

# Определяем модель User
class User(Base):
    __tablename__ = 'users' # Имя таблицы в базе данных

    # Колонки таблицы
    id = Column(Integer, primary_key=True, index=True)
    email = Column(String, unique=True, index=True)
    password = Column(String)
```

Замечания к второй лабораторной работе

Этапы выполнения:

1. Нужна вторая лабораторная работа, хотя бы её базовая часть (как её сделать описано выше, также можно взять мой проект, в коде достаточно комментариев);
2. Начинается самое интересное. Redislite не работает на Windows, поэтому будем ставить Docker на котором и развернем большой Redis. Ссылка на [Docker](#). Устанавливаем (нужна будет перезагрузка) и переходим во вкладку **Containers**, в поиске пишем **Redis** и нажимаем **Run**. В нижнем правом углу нажимаем на **Terminal** и переходим в папку своей лабы через команду `cd`, затем прописываем `docker run -d --name redis-container -p 6379:6379 redis` - контейнер с редисом работает.
3. Теперь нужно поднять сервер с **Celery** + **Redis**. Для этого в папке 'app.celery' создаём два файла:
 - `celery_app.py` - здесь мы задаём параметры сервера (ссылки на бэкэнд и брокер; как данные сервер принимает и как отправляет обратно; все задачи которые на нём запускаются)

- `tasks.py` - здесь описываются задачи, которые мы хотим запускать на сервере Celery

`celery_app.py` в моём случае выглядит следующим образом:

```
from app.core.config import Settings
from celery import Celery

# Эта канал для уведомлений
REDIS_BROKER = f"redis://{Settings.REDIS_HOST}:{Settings.REDIS_PORT}/0"
# Это канал где выполняются тяжелые процессы
REDIS_BACKEND = f"redis://{Settings.REDIS_HOST}:{Settings.REDIS_PORT}/1"

celery_app = Celery(
    "lab3",
    broker=REDIS_BROKER,
    backend=REDIS_BACKEND,
)

# Определяю то, в каком формате отправлять и принимать запросы
celery_app.conf.update(
    task_serializer="json",
    result_serializer="json",
    accept_content=["json"],
    result_expires=3600,
)

# Передаю все "тяжелые" процессы которые выполняются на celery + redis
celery_app.autodiscover_tasks(['app.celery.tasks'])
```

По стандарту хост Redis называется `localhost`, а портом тот, на котором запускается сервер Celery (то есть Docker-контейнер, смотри выше)

`tasks.py` в моём случае выглядит следующим образом:

```
# time нужно для имитации длительности процесса
# json нужен, чтобы обрабатывать запросы
import time, json
# Параметры сервера Celery
from app.celery.celery_app import celery_app
# Параметры сервера HTTP и Redis
from app.core.config import Settings

# В паме явно объявляю путь и имя "тяжелого" процесса
@celery_app.task(bind=True, name="app.celery.tasks.long_running_parse")
def long_running_parse(self, url: str):
    # Открываю соединение с каналом Redis
    from redis import Redis
    r = Redis(host=Settings.REDIS_HOST, port=Settings.REDIS_PORT, db=0)
    # Пушим уведомление
```

```

result = {"task_id": self.request.id, "status": "in progress"}
r.publish("notifications", json.dumps(result))
# пример "тяжёлой" операции
time.sleep(5)
# Пушим уведомление
result = {"task_id": self.request.id, "status": "done"}
r.publish("notifications", json.dumps(result))
return result

```

`notifications` - это тот самый канал `Websocket` который будет присылать уведомления клиентами о процессе запущенных ими задач, его определение я распишу далее, после того как разберемся с запуском `Celery` + `Redis`.

4. Для запуска `celery` используем `celery -A {дальше путь к файлу где объявляется celery в формате "app.celery.celery_app"} worker --loglevel=info --pool=solo`
5. Теперь, когда сервер `Celery` + `Redis` запущен (я надеюсь, потому что это может не случится и с 10 раз), нужно внести изменения в файл сервера FastAPI, в моём случае это `main.py`. Начнём с описания менеджера пользователей, подключающихся к серверу:

```

# Этот класс нужен для того, чтобы обслуживать сразу несколько клиентов
class ConnectionManager:
    def __init__(self):
        self.active: list[WebSocket] = []

    async def connect(self, ws: WebSocket):
        await ws.accept()
        self.active.append(ws)

    def disconnect(self, ws: WebSocket):
        self.active.remove(ws)

    async def broadcast(self, msg: dict):
        text = json.dumps(msg)
        for ws in self.active:
            await ws.send_text(text)

manager = ConnectionManager()

```

Теперь нам не важно количество подключенных к серверу, с этим разбирается `Websocket`.

6. Теперь нужно подключиться к `Celery` + `Redis`. Будем это делать при запуске сервера, что логично. За это отвечают следующие две функции:

```

# Подключение к Redis и запуск слушателя при запуске сервера
@app.on_event("startup")
async def on_startup():

```

```

# Глобальные переменные - зло, но голову ломать не хочу
# Объявляю подключение к redis глобальным, чтобы в дальнейшем можно было к
нему обращаться
global redis
# Слушаю канал для уведомлений
redis = await redis.from_url(REDIS_BROKER, decode_responses=True)
# запускаю в фоне цикл, который слушает канал
asyncio.create_task(notify_loop())

async def notify_loop():
    sub = redis.psubsub()
    # Указываю, что подписываюсь на канал websocket - notifications
    await sub.subscribe(WebsocketInfo.NOTIFICATIONS)
    while True:
        msg = await sub.get_message(ignore_subscribe_messages=True,
        timeout=None)
        if msg and msg["data"]:
            data = json.loads(msg["data"])
            await manager.broadcast(data)
            await asyncio.sleep(0.01) # не жрём 100% CPU

```

`on_startup()` при запуске сервера подключается к уже работающему серверу **Celery** + **Redis** и начинаем внимательно слушать что на нём происходит (конкретнее он следит за выполняемыми на нём задачами)

`notify_loop()` занимается этой самой прослушкой канала **Websocket**, причем внутри канал может быть разбит на произвольное количество подканалов, в моём случае это **notifications**. Он замечает все сообщения, разбирает и отправляет пользователям информативную часть.

7. Но к каналу **Websocket** тоже нужно подключиться, за это отвечает следующая функция:

```

# точка входа WebSocket
@app.websocket(f"/ws/{WebsocketInfo.NOTIFICATIONS}")
async def ws_notifications(ws: WebSocket):
    # Ждём подключения клиента
    await manager.connect(ws)
    try:
        while True:
            # Ожидая пинг от клиента
            await ws.receive_text()
    # Если клиент прервёт соединение, то закрываю канал для него
    except WebSocketDisconnect:
        manager.disconnect(ws)

```

8. Теперь можно определить новые или переопределить старые задачи для работы в **Celery** + **Redis** с отправкой уведомлений через **Websocket**. Это делается так:

```

# Тестовая функция для проверки связки Celery + Redis + REST API
@app.post("/tasks/parse")

```

```

async def run_parse(req: ParseRequest):
    task = long_running_parse.delay(req.url)
    return {"task_id": task.id}

```

Моя тестовая функция делает вид что парсит сайт. Здесь можно почитать про [delay](#), если кратко - эта функция получает уникальный id задачи, который нам пригодится в клиенте.

9. Теперь пришло время описать клиентскую сторону с регистрацией, авторизацией и выполнением долгих задач на стороне [Celery](#) + [Redis](#). Дело вкуса, я делал это через класс приложения. Полный код можно в [client.py](#) со всеми комментариями, но, во избежание ментальных травм разного рода, предлагаю посмотреть на него через мои пошаговые объяснения (с ними, надеюсь, этот бред будет более понятен). Сразу нужно заметить, что приложение целиком является асинхронным из-за того, что уведомления от Websocket выводятся асинхронно, как только обновляется статус задачи (обновление статуса задачи я не описал, но, фактически, это просто последовательный пуш в [Redis](#) json-файлов из пункта 3 в файле [tasks.py](#)).

10. Начнём с первого момента, который может вызвать вопрос - список команд и общий цикл клиентского приложения.

В словаре хранятся функции всех исполняемых команд, сам же словарь объявляется при инициализации объекта класса [Application](#).

```

# Список команд
self.commands: Dict[str, Callable] = {
    "login": self.login,
    "registr": self.register,
    "task": self.create_task,
    "cls": self.clear_console,
    "exit": self.exit,
}

```

Из библиотеки [typing](#) вызываю Dict просто на всякий случай (ну или потому что deepseek так написал, только тсссс)

Основной цикл приложения описан в функции [command_handler\(\)](#)

```

async def command_handler(self):
    while self.running:
        try:
            command = await self.session.prompt_async(
                "Введите команду (login/registr/task/cls/exit): ",
                refresh_interval=0.1
            )

            if command in self.commands:
                await self.commands[command]()

```

```

        else:
            await self.safe_print("Неизвестная команда")
            await self.safe_print("Доступные команды: " + ",
".join(self.commands))

        except KeyboardInterrupt:
            await self.exit()
        except Exception as e:
            await self.safe_print(f"Ошибка: {str(e)}")

```

`prompt_async()` из библиотеки `prompt_toolkit` используется для нормального ввода/вывода в асинхронном приложении, если попробовать написать что-то своё, то велик шанс того, что ожин ввод/вывод будет перекрывать другой ввод/вывод.

Видно, что пока приложение выполняется, функция ищет соответствующий вводу пользователя ключ в словаре `commands`, если таковой находится, то вызывается сама функция `await self.commands[command]()`, в противном случае выводится список доступных команд.

11. Теперь поговорим про слушатель `Websocket`. Не достаточно чтобы сервер `FastAPI` подключился к `Celery` + `Redis`, нужно чтобы сам клиент подключился к серверу и слушал что происходит в этом канале. Что происходит в этом канале? Вспоминаем про менеджер пользователей, когда на `Celery` + `Redis` выполняется "долгая" задача, уведомления об этом отправляются на сервер `FastAPI` и менеджер пользователей ловит все эти уведомления и отправляет в канал `Websocket` между `FastAPI` и клиентом. Код слушателя ниже:

```

async def websocket_listener(self):
    while self.running:
        try:
            # Подключение к каналу Websocket
            async with websockets.connect(self.ws_url) as ws:
                async for message in ws:
                    data = json.loads(message)
                    # Смотрю на id задачи уведомление которой получил
                    # если она совпадает с id какой-то активной задачи
                    # то вывожу уведомление и удаляю из id из активных задач
                    if data['task_id'] in self.active_tasks:
                        await self.show_notification(json.dumps(data,
ensure_ascii=False))
                        # Как только задача выполнена - она удаляется из списка
                        # активных задач пользователя
                        if data["status"] == "done":
                            self.active_tasks.remove(data['task_id'])

        except Exception as e:
            await self.safe_print(f"WebSocket error: {str(e)}")
            await asyncio.sleep(5)

```

Как видно, изначально каждый клиент получает все уведомления, даже тех задач, которые инициализирует не он. Это косяк, который я решил следующим образом: у каждого клиента есть свой список активных задач с их id и, при получении уведомления, мы смотрим на id задачи по которой получаем уведомление и либо выводим либо не выводим её, удаление же из активных задач происходит только когда статус задачи **done**.

12. Всё! Теперь можно посмотреть на весь остальной код и постараться осмыслить всё там написанное. Надеюсь теперь это будет полегче. В принципе, на этом всё - остальное делается по вариантам. Согласно варианту нужно закинуть задачи, которые раньше выполнялись на сервере в файл **tasks.py** (нужно будет немного переписать сами функции и не забыть добавить уведомления через **Websocket**) и написать в **main.py** (в файле сервера **FastAPI**) обработчики для этих задач, чтобы они запускались именно на стороне **Celery** + **Redis**.