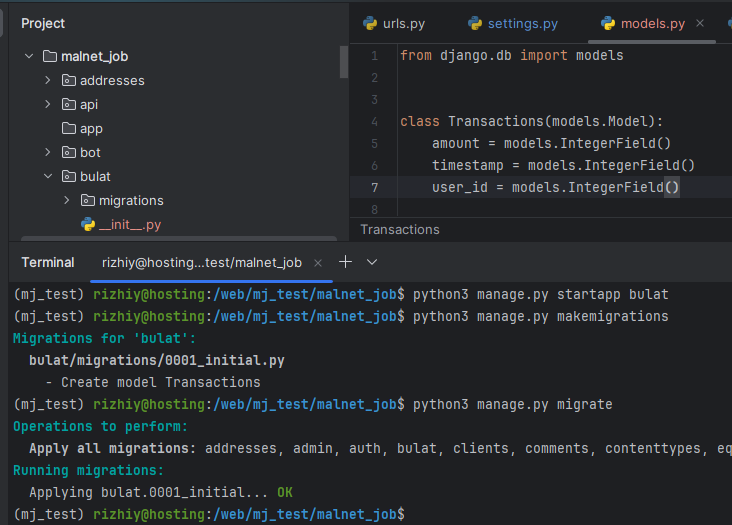
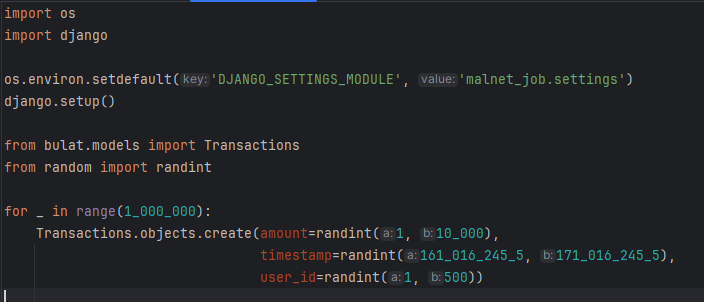
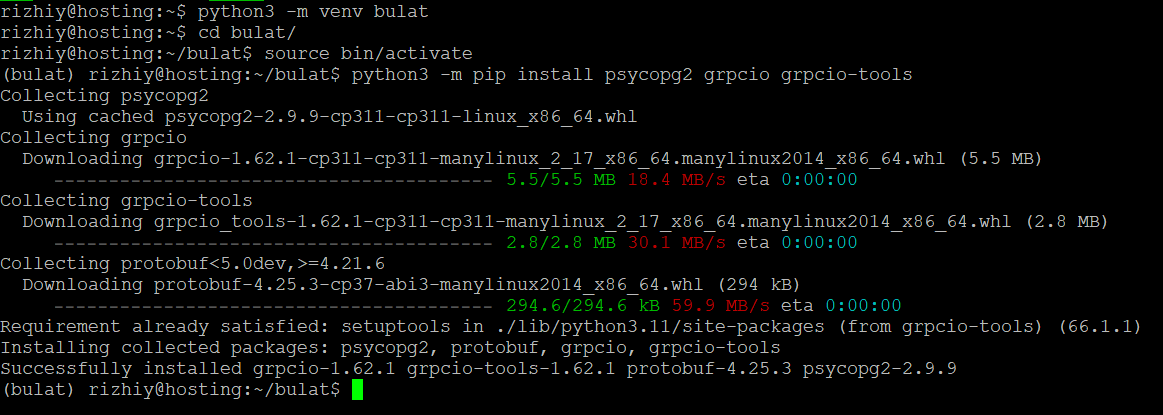
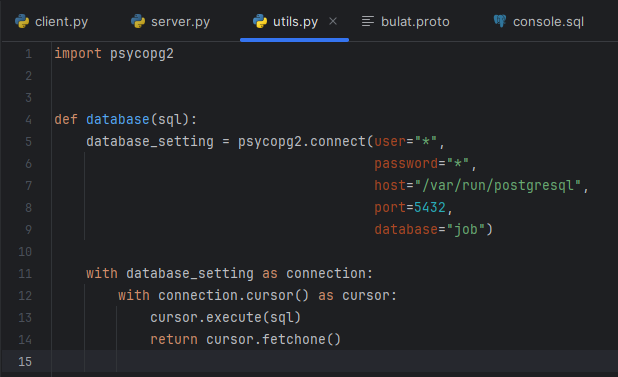
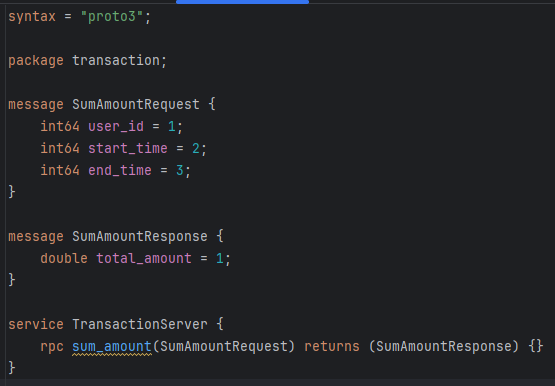
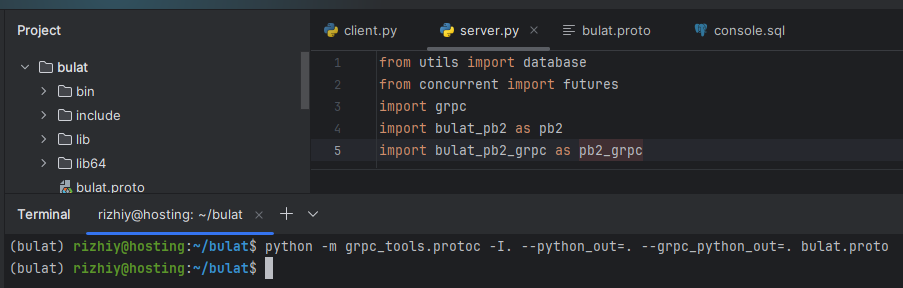
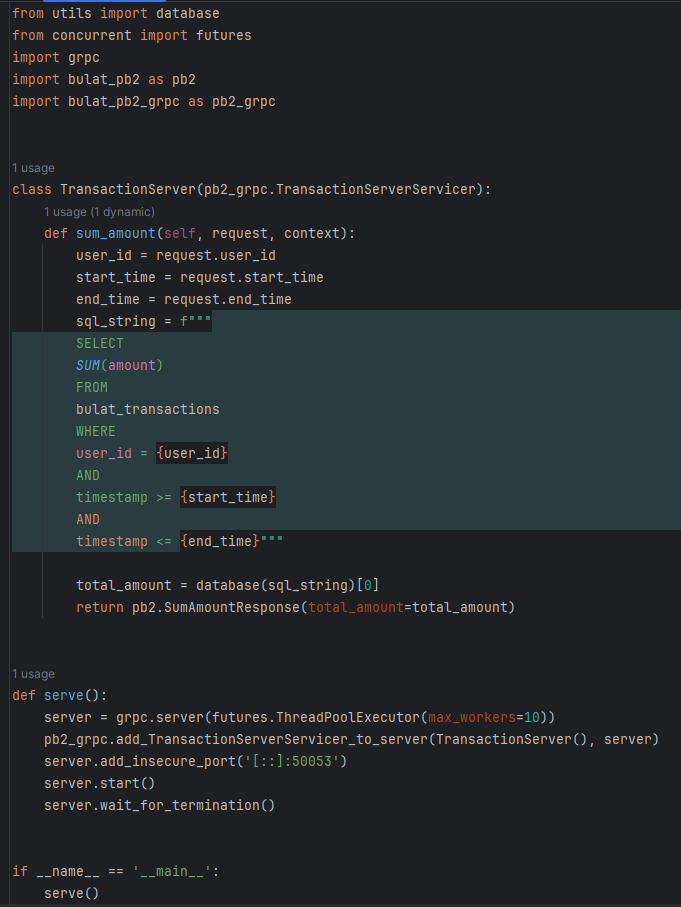
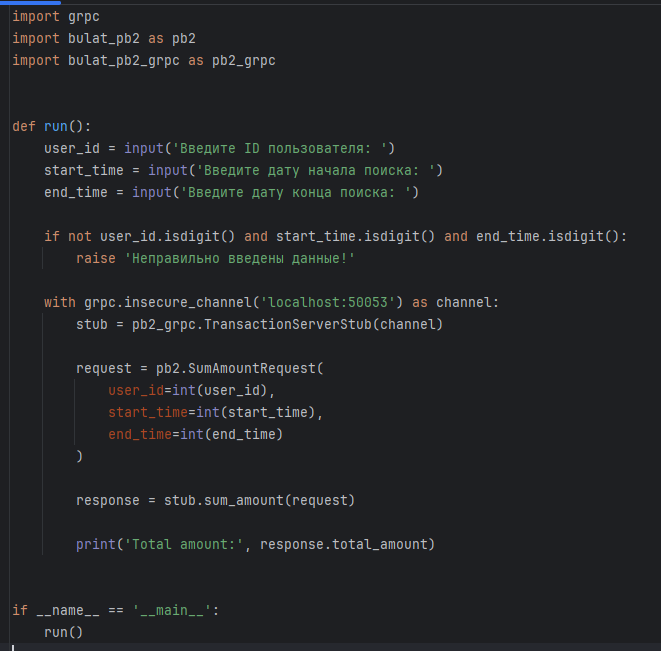
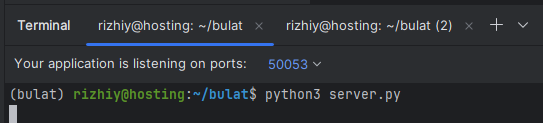
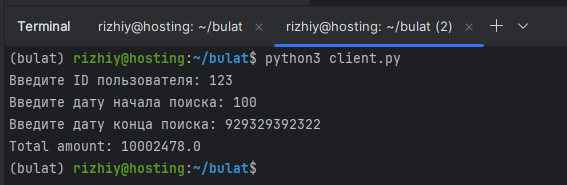
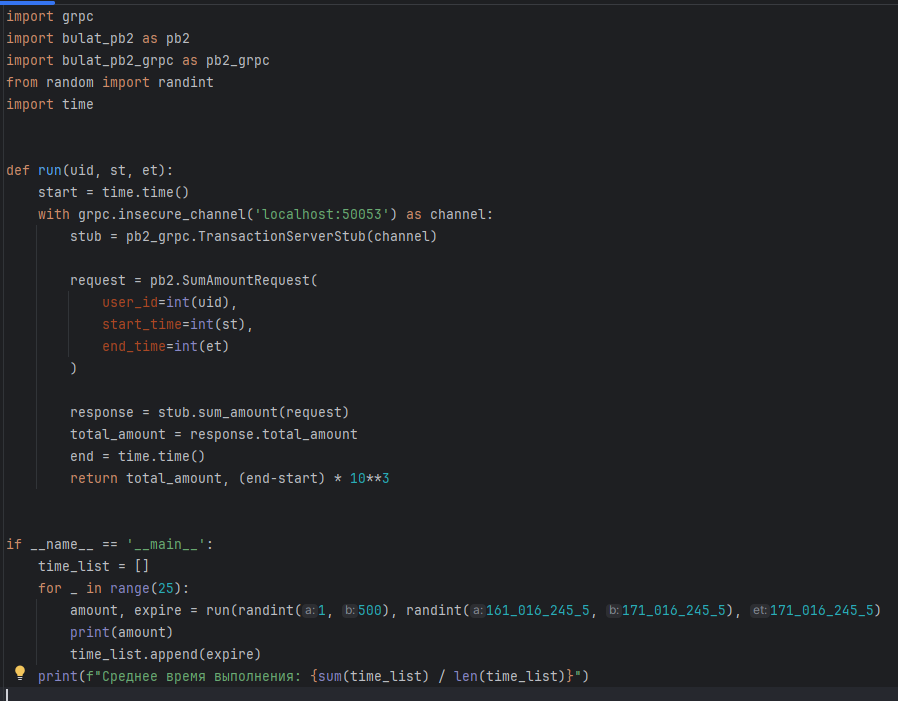
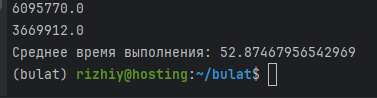
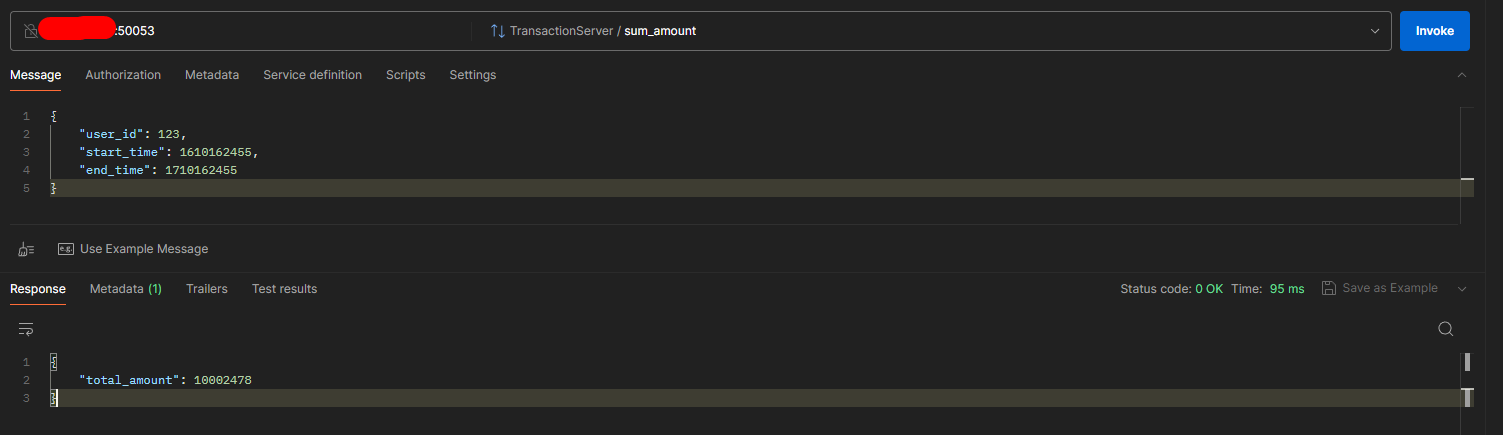
**Тестовое задание**

1. Создал модель базы данных. Во всех полях тип поля Int. Решил использовать Django ORM для создания. Тесты проведу через raw SQL и тот же ORM  
   
2. Создаём миллион записей. Сумма от 1 до 10000, время от 2021 года до сегодня, а пользователей, предположим, 500  
   
3. Пока создаются записи в базе данных, я создал виртуальную среду, где будет происходить развёртка gRPC и установил все необходимые зависимости. Я решил взять стандартную реализацию gRPC – ту, которую рекомендует официальный сайт  
   
4. После этого реализую «клиент-серверную» архитектуру. Создам 2 файла: 1 – отвечающий за отправку данных (server.py), другой – за принятие данных (client.py)
5. Помимо этого, приближая условия задания к реальным, создаю файл utils.py, где будет лежать функция обращения к базе данных. Обернул её в менеджеры контекста для большей надёжности  
   
6. Далее создаём файл bulat.proto для конфигурации. Так как на вход у нас будет user\_id, start\_time и end\_time, а на выходе только сумма, то описываем следующую структуру:  
     
   Я описал такую структуру под конкретную задачу, далее, при необходимости, её можно расширять с последующей пересборкой
7. После написания файла конфигурации проводим его сборку и импортируем 2 добавленных файла в файл сервера  
   
8. Далее приступаем к написанию кода сервера. В нём будет необходимо объявить 2 сущности: класс, определяющий данные, которые должны быть в ответе и функцию, которая запускает сервер gRPC. Также сделаем так, что этот файл можно запустить только при прямом доступе  
     
   
9. Переходим к написанию клиента. В целом, можно было выбрать любой интерфейс передачи данных, поэтому сделаем ввод с консоли. В реальных условиях это может быть запрос от gRPC, HTTP или иного интерфейса



1. Проводим тестирование. В одном терминале запускаем сервер:  
     
   А в другом клиент и вводим данные:  
   
2. Сервер и клиент работают. Теперь проведём замеры скорости. Для начала попробую поставить временные метки в коде клиента и взять среднее между 25 разными запросами, немного изменив код клиента  
    
3. Финальным этапом будет замер скорости через postman. Но перед этим я решил перенести код клиента на свой локальный компьютер и провести замер скорости таким же образом. Пришлось вынести сервер на внешний IP адрес для доступа извне. Среднее время выполнения было 63.60740661621094.
4. Для тестирования через postman указываем в нём сетевой сокет сервера, proto-файл и тело сообщения  
     
   Проведя 10 таких запросов, я измерил, что среднее время запроса – 110 мс. Разброс между значениями был от 90 до 140 мс, что скорее всего происходит из-за неоднородности сетевого подключения и установки нижнего порога для начала времени

**Вывод:** задание было весьма интересным. Радует то, что параллельно с тем, как я его делал, я смог понять суть и концепцию данной технологии и нашёл её очень полезной и удобной, что по скорости обращения, что по технической реализации. Ради эксперимента я попробовал выдать такие же данные по методу GET HTTP, но они оказались в 1.5-2 раза медленнее. Далее я попробую углубиться в этот протокол, чтобы более точно узнать, как он устроен и как с ним работать в разных средах исполнения

При выполнении задания трудностей не возникло, разве что непривычен формат файла «.proto». А так, большую часть времени у меня заняло создание 1000000 записей в базе данных

Также отмечу то, что я не оставлял комментарии в коде, ибо писал аннотацию к каждому своему шагу в данном файле