МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
(МОСКОВСКИЙ ПОЛИТЕХ)

Факультет информационных технологий
Кафедра «Инфокогнитивные технологии»

КУРСОВОЙ ПРОЕКТ

на тему: *«Разработка вспомогательного веб-сервиса для целевого поиска спортивных площадок в городе Москве»*

Направление подготовки 09.03.03 «Прикладная информатика»
Профиль «Корпоративные информационные системы»

Выполнил:

студент группы 211-362

Некрасов Олег Дмитриевич

25.01.2023 _____
(подпись)

Москва 2023

СОДЕРЖАНИЕ

Введение

Людям занимающимся спортом бывает тяжело найти подходящую для себя площадку. К примеру на онлайн-картах не всегда отображаются более новые площадки, как правило нет конкретных характеристик и можно потратить много времени до того момента как человек найдёт для себя оптимальный вариант практическим путём. Данный сервис разработан для помощи людям, не равнодушным к своему образу жизни и здоровью, с целью оптимизировать их время и помочь выбрать самый оптимальный вариант под конкретные задачи, предоставляя возможности поиска удобной площадки исходя из геопозиции, конкретных характеристик, а также выбора пользователя.

1 Цель и задачи работы

Целью настоящей работы является разработка веб-сервиса для поиска и рекомендаций спортивных площадок для пользователей.

В качестве датасета был выбран набор данных «Тренажерные городки (воркауты)» с официального сайта портала открытых данных правительства Москвы [5].

Для реализации поставленной цели, были выявлены следующие задачи:

1. Проектирование модульной и расширяемой системы.
2. Разработка рекомендательной системы на основе понравившихся пользователю площадок.
3. Разработка рекомендаций на основе поискового запроса пользователя.
4. Разработка рекомендаций на основе геопозиции пользователя.
5. Разработка команды администратора для обновления датасета.

6. Настройка интерактивной документации openapi.

Для разработки сервиса был выбраны следующие средства:

1. Язык программирования Python.

2. База данных Postgres, для хранения данных датасета, данных пользователей

3. База данных Redis, используется как брокер сообщений и для хранения ключей аутентификации.

4. Docker - для автоматизации развертывания на сервер.

5. WSGI сервер gunicorn, для репликации веб-процессов.

6. ASGI сервер uvicorn как асинхронный интерфейс для коммуникации между сервером и клиентом.

7. Веб-сервер Nginx — настройка проксирования запросов на WSGI-сервер для отображения документации и интерактивной отправки запросов.

2 Проектирование приложения

Разрабатываемая система позволяет:

1. Регистрироваться/аутентифицироваться пользователям.

2. Предоставлять клиентам данные о площадках в различных контекстах.

3. Добавить площадку в понравившиеся от имени пользователя.

4. Обновлять набор данных а также проводить необходимую предобработку в автоматическом режиме от имени администратора.

Диаграмма вариантов использования представлена на рисунке 1.

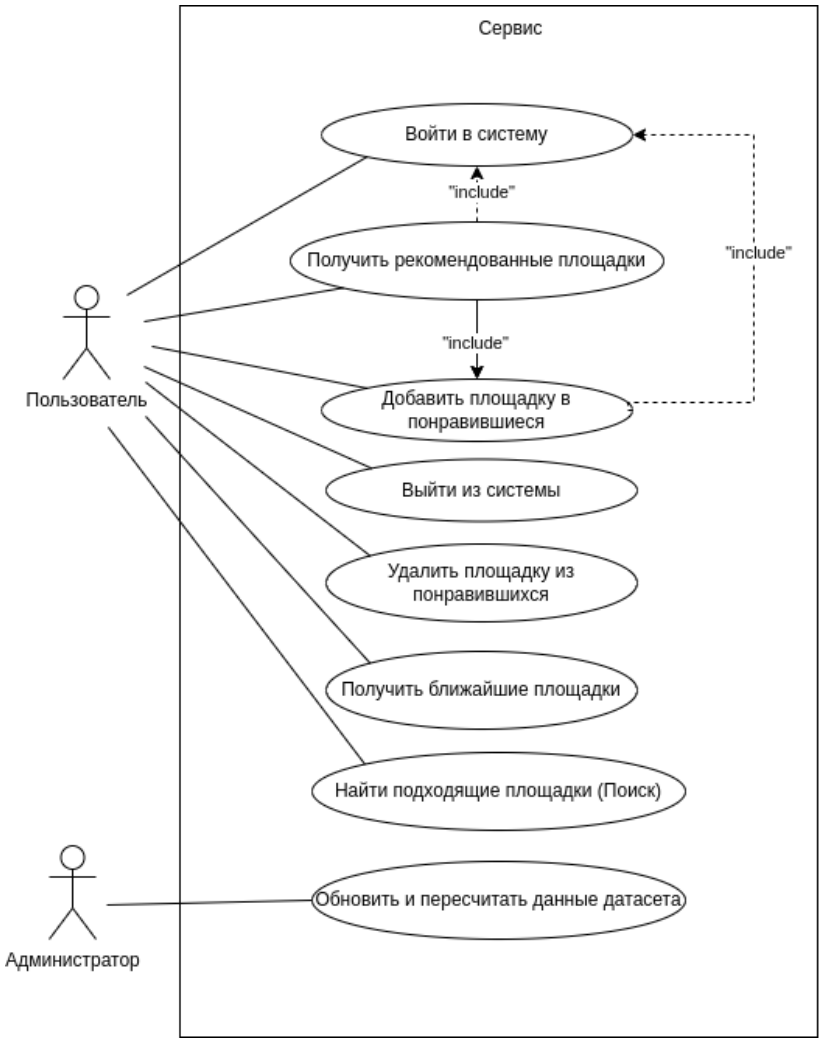

Рисунок 1 - Диаграмма вариантов использования

В первую очередь при проектировании приложения и анализа его функциональных возможностей было принято решение разделить ответственности на базовый процесс и вычислительный процесс. Исходя из того, что задачи по большей части используют математические методы обработки больших данных, которые постоянно расширяются, необходимо отделить данный спектр задач от основного потока выполнения программы.

Вычислительные задачи помещаются в очередь, которая грамотно использует вычислительные мощности машины, а именно: помещение задачи в очередь, распараллеливание задач на несколько процессов и доставка результата по запросу на базовый веб-процесс, каких тоже может быть

несколько. Для доставки сообщений с веб-процесса на вычислительные и обратно происходит с помощью брокера — Redis, данные предварительно сериализуются в json. Веб-процесс является асинхронным, и также поддеживает репликацию (увеличение производительности путём увеличения количества процессов). Плюсы такой структуры взаимодействия очевидны, иначе при выполнении синхронных вычислительных задач в основном потоке приложения приводило бы к застою веб-сервера и существенную потерю производительности.

Следующий шаг: проектирование внутренних компонентов API. Конкретно - выделение сущностей, их взаимодействия с бизнес-логикой и средствами инфраструктуры. Упрощённая схема с разделением на слои и их взаимодействия представлены на рисунке 2. (Упрощённая т.к основное внимание уделено слоям, а не классам). Стрелками указаны направления зависимостей.

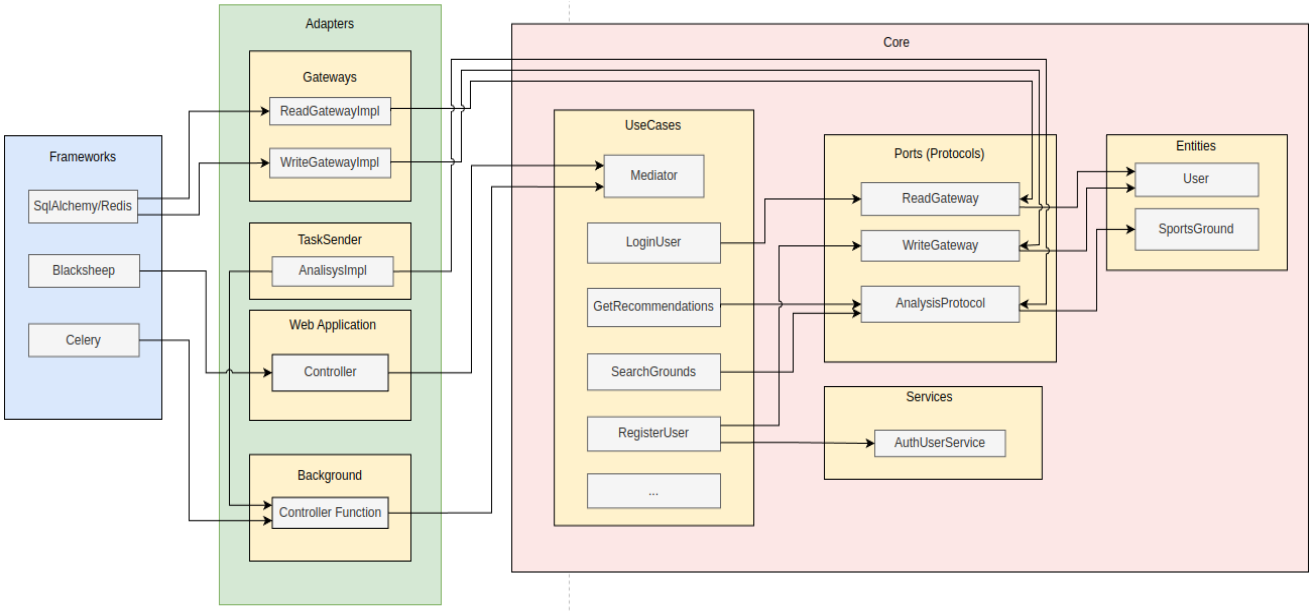

Рисунок 2 — Диаграмма взаимодействия слоёв приложения

Из рисунка видно базовое разделение слоев:

1. Слой фреймворков.
2. Слой адаптеров.
3. Корневой слой, который включает в себя бизнес-правила и их взаимодействие с сущностями (моделями предметной области), а также порты (протоколы) для взаимодействия с инфраструктурой (адаптерами).

Стоит обратить внимание что все зависимости направлены внутрь, что позволяет изолировать каждый внутренний компонент от внешнего.

Слой фреймворков — самый верхний слой, в нём не пишется много кода, в нём представлены внешние библиотеки.

Слой адаптеров отвечает за преобразование данных из формата сценариев и сущностей в форматы к примеру удобные для базы данных и/или для представления и наоборот. Здесь находятся классы которые реализуют протоколы от которых зависят сценарии.

Доменный/корневой слой, здесь находится вся бизнес-логика проекта, анемичные модели (entities), сценарии (usecases), протоколы/порты (ports/protocols), и вспомогательные сервисные классы (services). Сценарии зависят от протоколов и сервисов и представляют из себя конкретный кейс использования приложения, к примеру логин пользователя или поиск площадок, проще говоря сценарии это точка входа в приложение. Сущности в данном приложении соответствуют концепции анемичной модели, исходя из того, что они не содержат высокоуровневой бизнес-логики, они просто держат себя в валидном состоянии и используются сценариями. Порты/протоколы это абстракции над деталями реализации обращения к внешним сервисам (баз данных или брокеру сообщений).

Структура проекта представлена следующим образом:

1. core — соотносится со слоем ядра схемы.

2. infrastructure — соотносится со слоем адаптеров содержит блоки обращения к внешним сервисам Gateway — базы данных, и TaskSender — брокер сообщений.

3. presentation — соотносится со словем адаптеров и содержит блоки Web Application, Background, а также блок CLI который не вошёл в схему в целях экономии места.

4. resources — содержит константные значения в файле strings.py

5. settings.py — файл для работы с переменными окружения.

Основные плюсы такого разделения:

1. Решение проблемы расширения. Достигается это путём независимости бизнес-правил от деталей реализации конкретного фреймворка/библиотеки. К примеру: текущий веб-фреймворк Blacksheep легко заменяется Flask`ом или FastAPI. Т.е на одни и те же бизнес-правила могут использовать разные интерфейсы.

2. Лёгкость в тестировании, из-за того что вышестоящие слои легко заменяются имеется возможность заменить компоненты на какие-либо тестовые классы.

3. Обособленная доменная логика (core) помогает быстрее искать ошибки и неточности.

4. Решение проблемы внедрения нового функционала. При правильном построении можно не бояться за то, что при добавлении новой функциональности что-то может сломаться.

Однако у данного подхода есть и свои минусы:

1. Нужно писать больше кода.
2. Не выгодно в краткосрочной перспективе. Порог входа выше из-за того, что достаточно много времени в начале уделяется проектированию, написание какие либо абстракций и настройка взаимодействия отдельных модулей.

Рассмотрим структуру базы данных на примере схемы (рисунок 3).

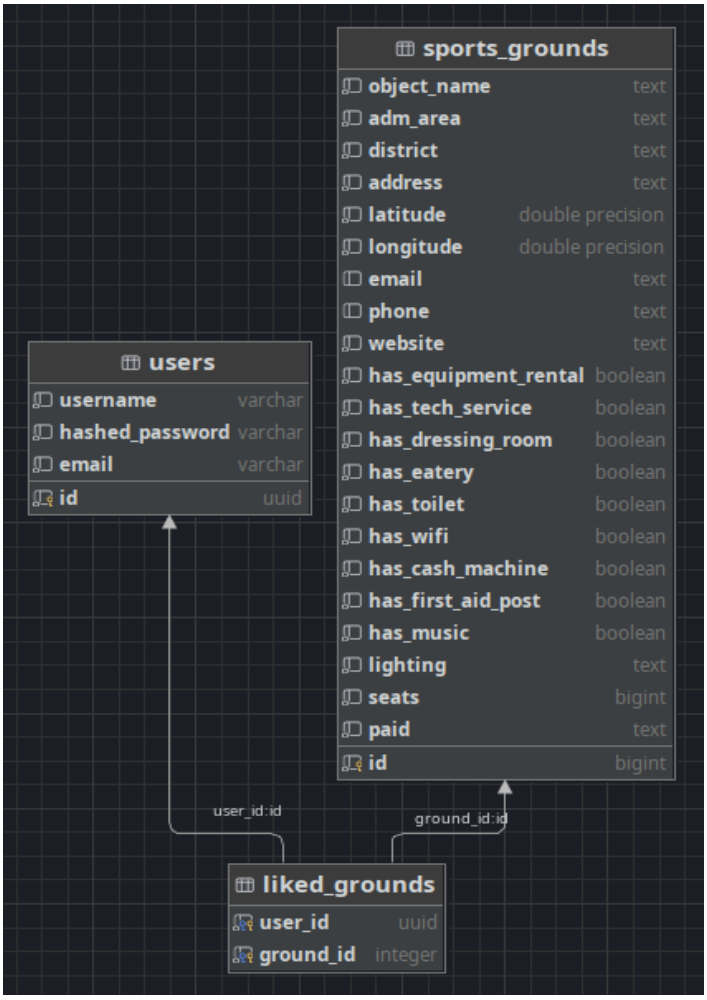

Рисунок 3 — Схема базы данных

Было выделено 3 сущности: пользователь, спортивная площадка, а также корзина площадок, которая содержит в себе понравившиеся пользователям площадки.

Рассмотрим импорт датасета в базу данных. Данный процесс является довольно затратным по ресурсам машины и запускается только администратором на стороне сервера через консоль. Импорт подразделяется на несколько основных частей:

1. Парсинг веб-страницы и выделение актуальной ссылки.
2. Загрузка и распаковка zip-архива в память.
3. Обработка датасета и сохранение в csv
4. Вычисление матрицы косинусной схожести и сохранение в csv.

5. Обработка типов данных и импорт в базу данных

Данная операция имеет как I/O так и CPUBound нагрузку. Чтобы не блокировать асинхронный loop данная задача помещается в отдельный процесс. Статус обновления датасета по каждой части выводится в логи.

Рассмотрим математические методы, которые позволяют извлекать из данных что-либо полезное. В данном приложении реализуются следующие вычислительные задачи:

1. Поиск ближайших площадок исходя из геопозиции пользователя
2. Рекомендательная система на основе лайкнутых пользователем площадок.
3. Рекомендательный поиск по 16 параметрам.

Поиск ближайших площадок, происходит путём реализации алгоритма метрического дерева. При обработке датасета изначальные координаты (latitude, longitude) преобразуются в радианы, так как для расчёта рассчёта расстояния между точками используется гаверсинус. Гаверсинусное расстояние точнее, если точки расположены далеко друг от друга, чем декартово расстояние. Происходит кластеризация исходных точек и заполнение бинарного дерева. Для начала выбирается самая центральная точка относительно всех, строится окружность где радиус это расстояние от центра до самой удалённой точки. Далее уже внутри этого шара выбирается 2 самые удалённые точки и также выстраивается 2 шара, и так далее пока не достигается определённая заранее заданная вложенность. Таким образом образуется структура, которая состоит из вложенных шаров. Шары это кластеры в центре которых находятся центроиды. Центроиды называются вершинами дерева. Поиск ближайших соседей осуществляется путём обхода дерева начиная с корня, если расстояние до текущего узла больше порога то ветвь отбрасывается, иначе рассматривается дальше. Поиск заканчивается, когда все точки рассмотрены и выдаётся результующее решение.

2-3 алгоритмы похожи, потому что оба используют меру косинусной схожести, но по разному. Во втором случае данная мера расчитываетсся между всеми объектами. Расчитывается мера схожести каждого вектора с каждым, и нахождение похожих площадок сводится к прочтению заранее просчитанного файла и обхода матрицы с целью поиска площадок с максимальным коэффицентом. В третьем же случае конструируется полностью новый объект для которого будет строится матрица, так как пользователи вводят разные данные и конструируются постоянно разные объекты, коэффиценты схожести не просчитываются заранее как во втором случае. Векторы получаются путём группировки колонок и их значений в одну строку для каждой площадки, после чего данная строка преобразуется в вектор. Далее рассчитывается коэффицент схожести двух векторов, значение которого в пределах от нуля до единицы. После чего происходит сортировка и вывод первых самых похожих площадок (количество задаётся клиентом).

3 Реализация приложения

Рассмотрим сценарий работы приложения на примере UML диаграммы классов (приложение 1) и схемы взаимодействия компонентов (рисунок 4).

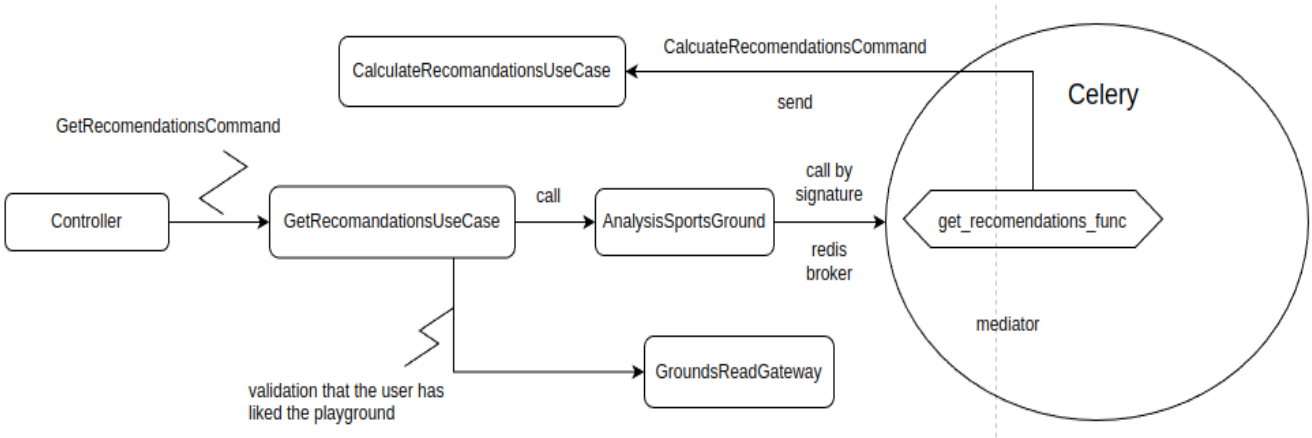

Рисунок 4 — Схема «получение рекомендаций»

Для удобства восприятия разделим данный сценарий на несколько частей: передача данных от контроллера к сценарию, взаимодействие сценария с инфраструктурой, отправка задачи в очередь, возвращение результата. Вызовы инфраструктуры со стороны сценариев будет рассматриваться на примере протоколов, механизм подстановки и развязывания зависимостей будет рассмотрен после описания сценария.

Для разделения бизнес-сценариев (usecase) на отдельные независимые модули используется паттерн Посредник (mediator). Контроллер отправляет команду на медиатор, после чего медиатор сопоставляет команде сценарий GetRecommendationsUseCase, и вызывает его метод handle, попутно передавая в аргументы команду, которая содержит необходимые для работы данные. Важно заметить что контроллер отвечает только за логику передачи информации в сценарий, что позволяет ему оставаться тонким. Также соблюдается принцип SRP (принцип единственной ответсвенности) для каждого сценария.

Класс сценария же выполняет валидацию, получает указанную пользователем площадку (entity) из базы данных с помощью класса GroundsReadGateway, формирует и отправляет задачу в очередь с помощью класса AnalysisSportsGround. Важно заметить, что валидация и получения сущности из базы данных происходит в рамках одной транзакции, открытие и закрытие транзакции отвечает класс UnitOfWork, после завершения транзакции сеанс с базой данных завершается.

После отправки задачи в очередь, часть приложения которая отвечает за распараллеливание вычислений принимает задачу от брокера и начинает её выполнение (реализация протокола AnalysisSportsGround вызывает по сигнатуре функцию get_recommendations). На стороне вычислений, функции которые представляют задачи также имеют зависимости от медиатора, который отправляет команду с данными о датасете на обработчик CalculateRecommendationUseCase, после выполнения результат вычислений

сохраняется у брокера. Важно заметить, что после отправки задачи в очередь сценарий GetRecommendationsUseCase заканчивает свою работу, возвращая идентификатор отправленной задачи. За получение обновлений по задаче отвечает отдельный сценарий, который обращается к брокеру и получает текущий статус. Похожим образом выполняются и другие вычислительные процессы: поиск подходящей площадки по параметрам, поиск ближайшей площадки по географическим координатам.

Посмотрев на диаграмму классов, можно заметить, что для получения рекомендованных площадок нужен конкретный пользователь. Сущность пользователя приходит в контроллер после его аутентификации. Проверка данных происходит в отдельном сценарии LoginUserUseCase который с помощью шлюза UserReadGateway и сервиса авторизации AuthUserService сверяет данные полученные от пользователя и фактические данные, которые находятся в базе данных, и возвращает идентификатор сессии. Данные о сессиях и их владельцах хранятся в базе данных Redis в зашифрованном виде и имеют определённый срок жизни, по истечению которого пользователю придётся заново войти в систему. Схема процесса аутентификации представлена на рисунке 5.

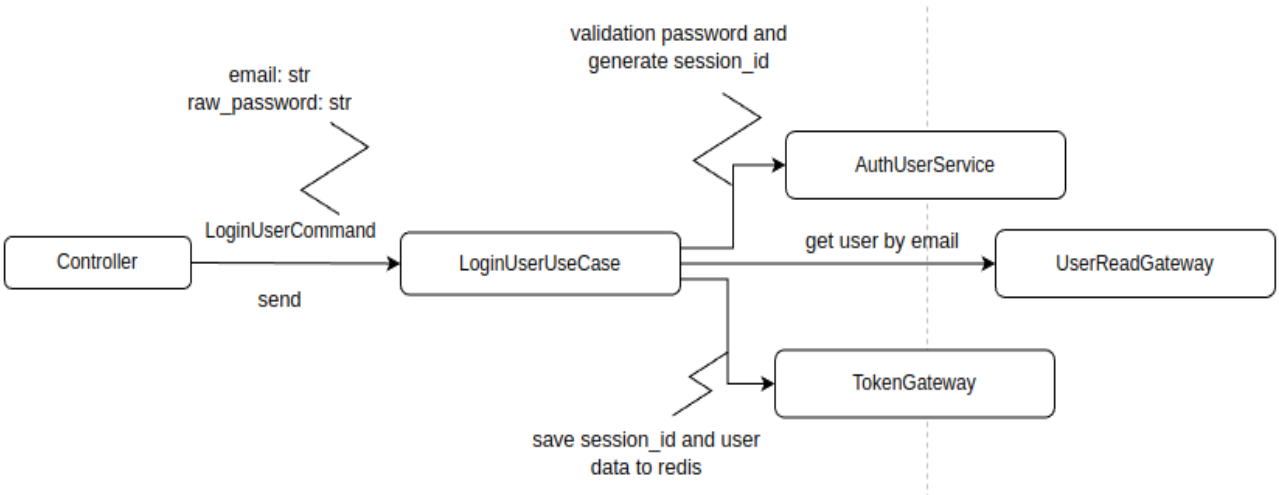

Рисунок 5 — Аутентификация пользователя

Рассмотрим механизм развязки зависимостей. Так как код строится по принципу DIP (принцип инверсии зависимостей), необходимо разрешать эти зависимости и подставлять нужные классы. Для развязки зависимостей используется механизм DI (внедрение зависимостей), веб-фреймворк Blacksheep предоставляет библиотеку rodi. Rodi регулирует процесс предоставления внешней зависимости, предоставляя контейнер в который мы помещаем конкретные классы или функции-фабрики, которые собирают конкретные классы. Также с помощью библиотеки можно регулировать срок службы зависимостей. Rodi предоставляет следующие варианты жизненных циклов:

1. Singleton — создание только одного объекта. Пример: класс формирующий документацию.

2. Transient — создание каждый раз, когда требуется. Пример: класс настроек проекта.

3. Scoped — создание один раз на каждый веб запрос. Пример: сессия базы данных, и все классы которые её используют.

Сборка и настройка DI контейнера происходит при старте приложения, код выполняющий сборку представлен на рисунке 6.

```python
def _setup_di(self):
    self._app.services.add_instance(self._routes_registry, RoutesRegistry)
    self._app.services.add_exact_transient(Settings)
    self._app.services.add_singleton_by_factory(openapi_docs_factory)
    self._app.services.add_scoped_by_factory(mediator_factory)
    self._app.services.add_scoped_by_factory(celery_factory)

    # Databases
    self._app.services.add_scoped_by_factory(sa_engine_factory)
    self._app.services.add_scoped_by_factory(sa_sessionmaker_factory)
    self._app.services.add_scoped_by_factory(sa_asyncsession_factory)
    self._app.services.add_scoped_by_factory(redis_client_factory)
    self._app.services.add_scoped(UnitOfWork, UnitOfWorkImpl)

    # Auth and users
    self._app.services.add_scoped_by_factory(auth_user_service_factory)
    self._app.services.add_scoped(UserReadGateway, UserReadGatewayImpl)
    self._app.services.add_scoped(UserWriteGateway, UserWriteGatewayImpl)
    self._app.services.add_scoped(TokenGateway, TokenGatewayImpl)

    # Workout
    self._app.services.add_scoped(
        AnalysisSportsGround, AnalysisSportsGroundImpl
    )
    self._app.services.add_scoped(GroundReadGateway, GroundReadGatewayImpl)
    self._app.services.add_scoped(
        GroundWriteGateway, GroundWriteGatewayImpl
    )
```

Рисунок 6 — Настройка DI при старте веб-приложения

Для асинхронного выполнения CPU Bound задач используется библиотека Celery, зависимости от медиатора настраиваются также при старте приложения, однако без помощи rodi, а вручную. Код реализации DI на уровне вычислителей представлен на рисунке 7.

```python
def build_celery() -> Celery:
    settings = Settings()
    app = Celery(
        "app",
        backend=settings.redis.redis_url,
        broker=settings.redis.redis_url,
    )
    mediator = build_mediator(settings)
    get_nearest_grounds_task = _inject_dependency_to_task(
        get_nearest_grounds, mediator=mediator
    )
    search_grounds_task = _inject_dependency_to_task(
        search_grounds, mediator=mediator
    )
    get_recommendations_task = _inject_dependency_to_task(
        get_recommendations, mediator=mediator
    )

    app.task(get_nearest_grounds_task, name="get_nearest_grounds")
    app.task(search_grounds_task, name="search_grounds")
    app.task(get_recommendations_task, name="get_recommendations")
    app.task(update_dataset, name="update_dataset")
    return app


def _inject_dependency_to_task(task: Callable, **depends) -> Callable:
    task_with_dependency = lambda *args: task(*args, **depends)  # noqa
    task_with_dependency.__name__ = task.__name__
    return task_with_dependency
```

Рисунок 7 — Настройка DI на вычислительном уровне

4 Основные сценарии использования приложения

Маршруты API можно разделить на 2 части: с участием пользователя, и общедоступные.

Рассмотрим маршуты пользователей.

Регистрация происходит по маршруту `/api/v1/users`. Отправляется запрос методом POST, в тело запроса помещаются данные представленные в листинге 1.

Листинг 1 — Тело запроса регистрации

```
{
  "email": "example@mail.com",
  "username": "example",
  "password": "Example_Password123"
}
```

В случае успеха придёт ответ с кодом 200 и текстом: «User created». В случае если пароль не надёжен, или введёт некорректный email/username, будут выдаваться ответы с кодом 403 и детальной информацией, что пошло не так.

Аутентификация пользователя происходит по маршруту: `/api/v1/auth/login`. Отправляется запрос методом POST, в тело запроса помещаются данные пользователя (листинг 2), также указывается MIME тип - application/x-www-form-urlencoded или multipart/form-data.

Листинг 2 — Тело запроса аутентификации

```
{
  "username": "example@mail.com",
  "password": "Example_Password123"
}
```

В ответ приходит данные json с кодом 200 (листинг 3).

Листинг 3 — Ответ запроса на аутентификацию

```
{
  "session_id": "0f5baf05-5b8c-425f-8233-4b721efa0f34"
}
```

Для того, чтобы выйти из системы, клиенту нужно отправить запрос с методом POST на маршрут `/api/v1/auth/logout`. В параметры запроса передаётся ключ session_id и значение — идентификатор сессии. К примеру

маршрут будет выглядеть так: `/api/v1/auth/logout?session_id=af0bd1d5-e2ee-452b-92b3-a1e4cd21c518`

В ответ на запрос приходят данные с кодом 200 (листинг 4). В случае некорректно введённого идентификатора возвращается ответ с кодом 404 и информацией: "Session not found"

Листинг 4 — Ответ на запрос о выходе пользователя из системы

```
{
  "detail": "User 19d8c758-3a57-465a-9108-789e5585995e logged
out"
}
```

Рассмотрим маршруты для взаимодействия с площадками.

Добавить в понравившиеся площадку можно по маршруту `/api/v1/grounds/like`. Отправляется метод POST с параметром запроса ground_id — идентификатор площадки и заголовком Authorization (листинг 5).

Листинг 5 — Параметр загловка Authorization

```
{
  «Authorization»: af0bd1d5-e2ee-452b-92b3-a1e4cd21c518
}
```

При верно введённых данных будет выдан следующий ответ с кодом 200, и текстом "«Ground likef by user {user_id}".

Для того, чтобы убрать лайк, достаточно выполнить точно такой же запрос только методом DELETE.

Для того, чтобы просмотреть список понравившихся пользователю площадок, нужно отправить GET запрос с заголовком авторизации на маршрут `/api/v1/grounds/my`.

После того как пользователь добавил площадку в список понравившихся, открывается доступ к маршруту для просмотра рекомендаций: `/api/v1/grounds/recommendations`. Отправка запроса происходит методом GET. В параметры запроса передаётся идентификатор площадки под ключом ground_id. Ответ в случае успеха представлен в листинге 6. Тип идентификатора говорит о том, какой тип ответа в поле «data». Тип статуса говорит о статусе выполнения задачи.

Листинг 6 — Успешный ответ на запрос о получении рекомендаций

```
{
     "type": "ID",
     "status": "PENDING",
     "data": "92a877b1-b657-4d0e-a749-9f4f244fdca2"
}
```

Чтобы получить обновление и проверить выполнена ли задача, нужно отправить запрос с методом GET на маршрут `/api/v1/grounds/get-updates/{task_id}`. В параметр пути передаётся идентификатор задачи — task_id. Успешный ответ от сервиса представлен в листинге 7.

Листинг 7 — Успешный ответ на запрос о получении обновлений по задаче

```
{
      "type": "DATA",
      "status": "SUCCESS",
      "data": [
         {
          "id": 1047342252,
          "object_name": ...,
          "location": {…},
          "contact":{...},
          "conditions": {…}
         },
      ]
}
```

Рассмотрим общедоступные (без аутентификации) маршруты.

Для получения ближайших площадок отправляется запрос методом GET на маршрут `/api/v1/ground/nearest`. В параметры запроса передаются координаты - latitude и longitude а также количество площадок-count. В ответ приходит идентификатор задачи (листинг 6).

Для поиска площадок отправляется запрос на маршрут `/api/v1/ground/search`. В параметры запроса передаётся количество площадок — count. В тело запроса передаются параметры площадки (листинг 8). В ответ приходит идентификатор задачи (листинг 6).

Листинг 8 — Тело запроса поиска площадок

```
{
     "object_name": "string",
     "adm_area": "string",
     "district": "string",
      "address": "string",
      …
}
```

Посмотреть данную информацию в интерактивном виде можно по ссылке: https://nekrasov-oleg.ru/workout/api/docs. Там же указаны возможные типы входных и выходных параметров. Однако маршруты, которые используют заголовки авторизации следует тестировать в отдельной программе, например в Postman.

Также основные методы API были протестированы в полном цикле взаимодействия. Статистика прохождения тестов представлена на рисунке 8.

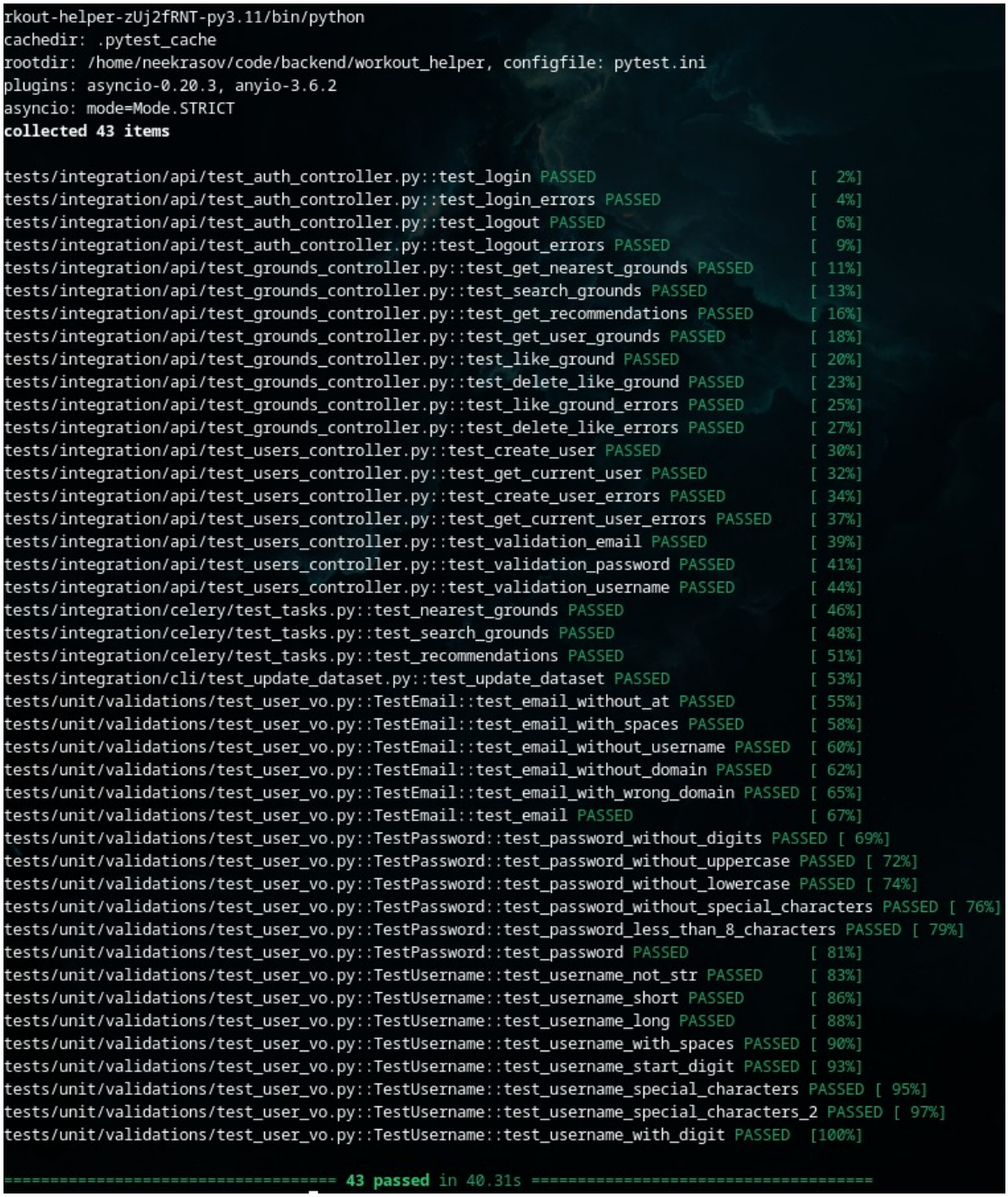

Рисунок 8 — Тестирование

Заключение

В результате данной курсовой работы было разработано доступное в интернете веб-приложение для рекомендаций и поиска спортивных площадок. На рынке уже существуют приложения с похожей предметной

областью: программы тренировок, сбор людей для общих тренировок и т.д. . Однако из альтернатив текущего приложения имеются только онлайн-карты, которые не предоставляют такой функционал. В дополнение можно улучшить рекомендательный алгоритм, добавив зависимость от оценок других пользователей, также в качестве расширения можно улучшить алгоритм поиска ближайших площадок по координатам, добавив возможность выдачи не только конкретной площадки и ей координат, но и расстояние от человека до площадки и оптимальный маршрут.

Ссылка на веб-приложение: https://nekrasov-oleg.ru/workout/api/docs.

Ссылка на исходный код: https://github.com/neekrasov/workout_helper.

Список литературы и интернет-ресурсов

1. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. / Р. Мартин. (дата обращения: 22.10.2022)

2. Документация по Python [Электронный ресурс]. URL: https://www.python.org/doc/ (дата обращения: 22.10.2022)

3. Документация по SqlAlchemy [Электронный ресурс]. URL: https://docs.sqlalchemy.org/en/14/ (дата обращения: 22.10.2022)

4. Документация по Blacksheep [Электронный ресурс]. URL: https://www.neoteroi.dev/blacksheep/ (дата обращения: 22.10.2022)

5. Набор данный «Тренажёрные городки (воркауты)» [Электронный ресурс] . URL : https://data.mos.ru/opendata/7708308010-trenajernye-gorodki-vorkauty/ (дата обращения: 22.10.2022)

Приложение 1

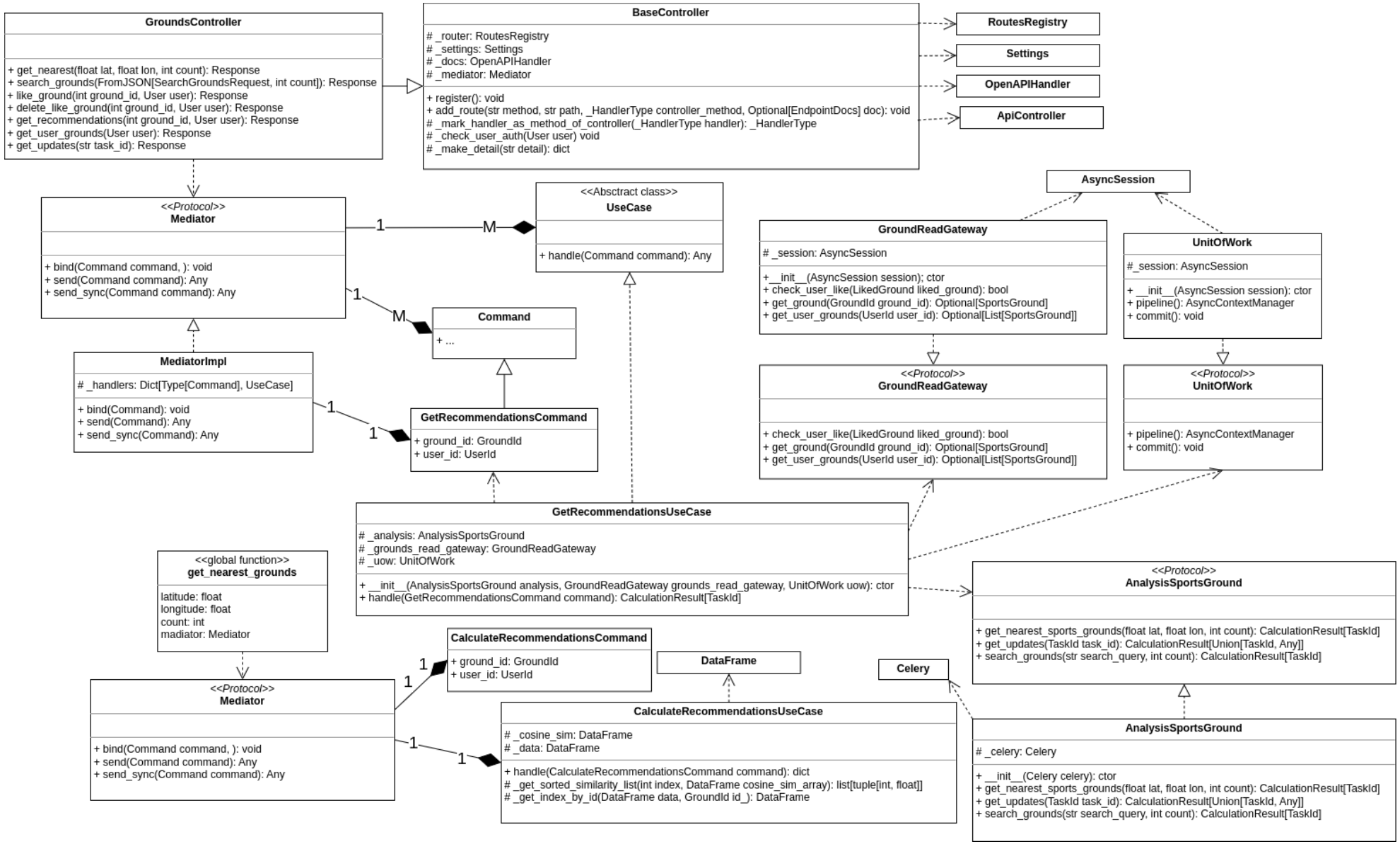

Рисунок 9-Диаграмма классов сценария "получение рекомендаций"