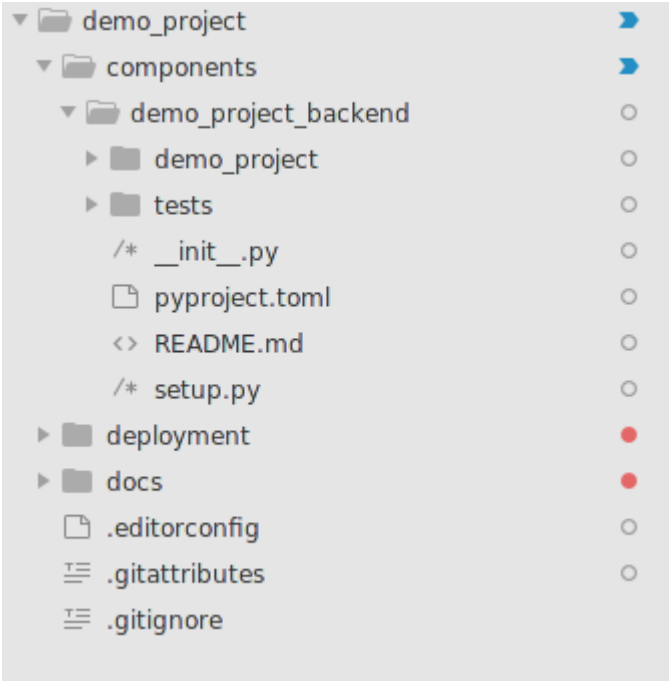
Требования по оформлению Python проектов

Автор последнего обновления: Anton Safronov | 24 янв. 2023 г.в10:47 GMT+5

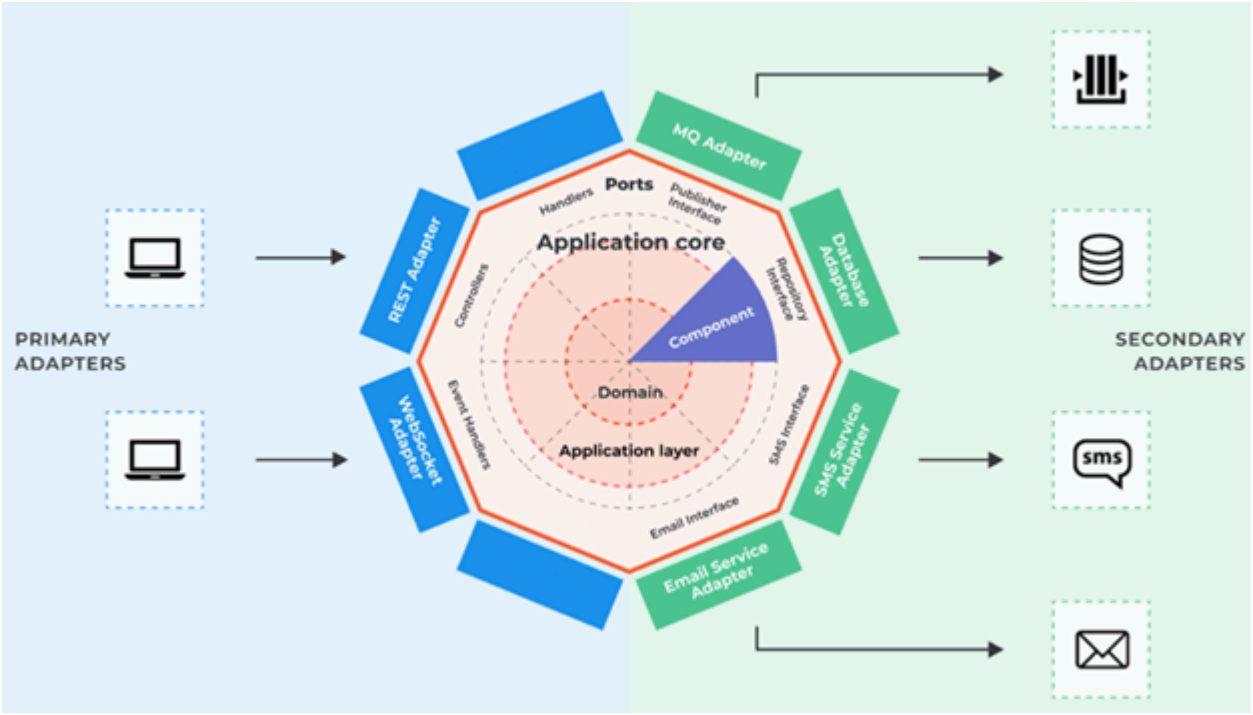
Требования по оформлению Python проектов



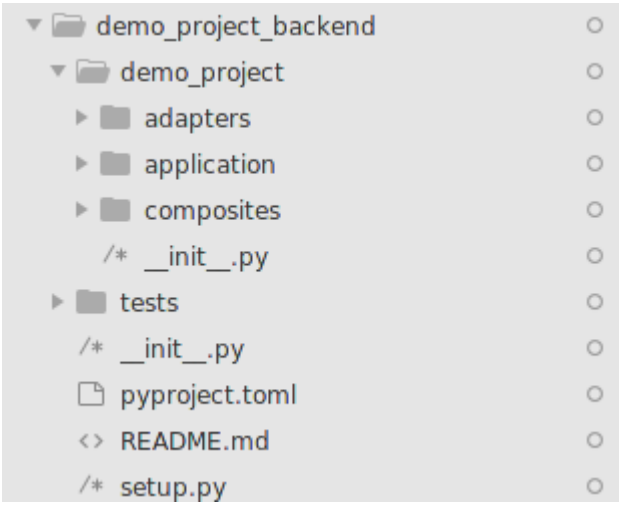
- используется монорепозиторий (см. в демо проекте)
  - в корне лежит .gitignore
  - так же в корне лежит .editorconfig
  - так же в корне лежит .gitattributes
  - в deployment лежат файлы для CI/CD (структуру обсудить с нашими девопсами)
  - в docs храним тех. документацию (схемы строим с помощью PlantUml)
    - схема прецедентов
    - схема базы данных
    - схема развертывания
    - схема компонентов
    - дока Swagger генерируется на беке при вызове соответствующего эндпоинта (см. демо проект)
    - **документацию к бизнес процессам держать либо тут, либо в отдельной wiki**
  - внутри лежит каталог "components", нужен чтобы разделить фронт и бек (структуру фронта обсудить с нашим фронтом)
  - в "components" лежит каталог "demo_project_backend", это корневой каталог для бека, этот же каталог является корневым для python модулей проекта, например в IDE можно его пометить как sources_root (в случае запуска из консоли, нужно определить PYTHONPATH и сослать на этот каталог)
- каталог бека оформляется в виде стандартного python пакета
  - в setup.py описываются метаданные пакета и зависимости
    - допускается использование setup.cfg
  - в pyproject.toml описываются различные конфиги сборщиков, автоформаттеров и тд. и тп.
  - `README.md` с кратким описанием проекта, указаниями как развернуть на локальной машине/контейнере, как запустить тесты, схема прав/групп и т.д.
  - каталог с исходным кодом - корень импортов внутри проекта, тут желательно использовать лаконичное имя

Демо проект находится в этом репозитории

Архитектура



Используется "Гексагональная" архитектура, первичные и вторичные адаптеры у нас в одном каталоге



Используется python 3.7
Наш стек:

- evraz-classic-app-layer
- evraz-classic-aspects
- evraz-classic-components
- evraz-classic-http-api
- evraz-classic-http-auth
- evraz-classic-messaging
- evraz-classic-messaging-kombu
- evraz-classic-sql-storage

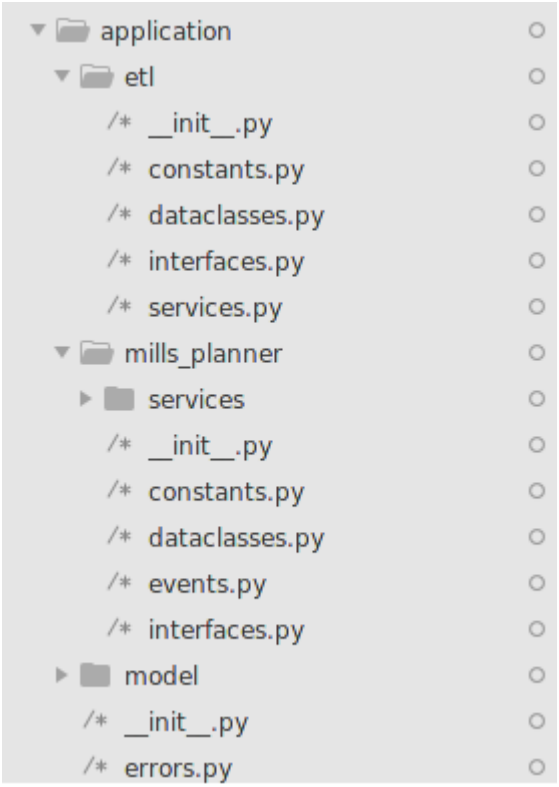
Брать самые свежие версии

Какие пакеты применяем для разработки:

- falcon~=3.0.0
- gunicorn~=20.0.0

- gevent~=21.1.0
- attrs~=21.2.0
- sqlalchemy~=1.4.0
- alembic~=1.7.0
- kafka-python~=2.0.0
- click~=7.1.0 (CLI)
- numpy~=1.21.0
- pandas~=1.3.0
- openpyxl~=3.0.0
- pydantic~=1.8.0
- pymssql~=2.2.0
- cx-oracle~=8.2.0
- kombu~=5.1.0
- psycopg2~=2.9.0
- PyJWT~=2.0.0
- python-json-logger~=2.0.0
- requests~=2.27.0
- plotly~=5.5.0
- pytest~=6.2.0
- pytest-cov~=2.12.0
- isort~=5.10.0
- yapf~=0.32.0
- toml~=0.10.2
- docxtpl~=0.16.4 Use a docx as a jinja2 template
- **остальные пакеты обсуждать с нашим лидером направления backend разработки**

Слой приложения (бизнес логика)



В слое приложения лежит все что относится к бизнес логике (сущности, DTO, константы, DS модель, сервисы и тд. и тп). Этот слой не зависит от интеграций (адаптеров). Для этого применяется механизм DI. В слое приложения описываются интерфейсы получения данных, в адаптерах они реализуются.

Если предметная область сложная, то создаются сущности предметной области (которые маппятся на таблицы в слое адаптеров). В сущностях описывается их поведение и инварианты.

Если предметная область простая (например, строим простые графики), то достаточно описать data transfer objects (DTO) и возвращать их из репозиториев/шлюзов/api клиентов в первичных адаптерах. Простые структуры данных (словари, списки скалярных значений, списки словарей) между слоями передаваться не должны. DTO это обычные датаклассы с простейшим поведением.

В сервисы внедряются вторичные адаптеры, и они работают с сущностями. Валидация данных происходит при вызове публичного метода сервиса из адаптеров, для этого на уровне сервисов описываются DTO в виде pydantic моделей.

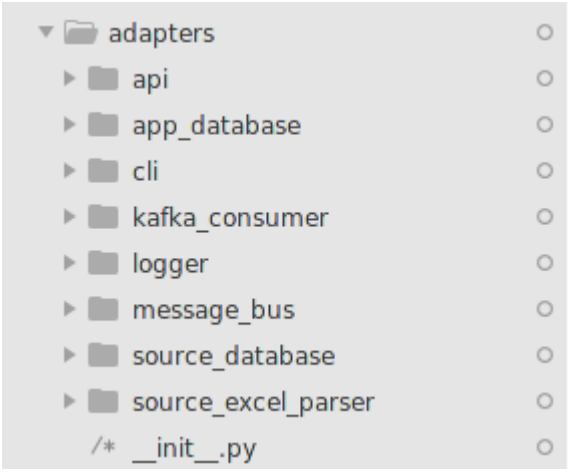
Из одного сервисы вполне можно вызывать другой. Тут вопрос проектирования предметной области, лучше сильно не запутывать связями.

Код DS (feature engineering и вызов моделей) лучше держать в отдельном пакете.

В этом же слое описываются всевозможные ошибки.

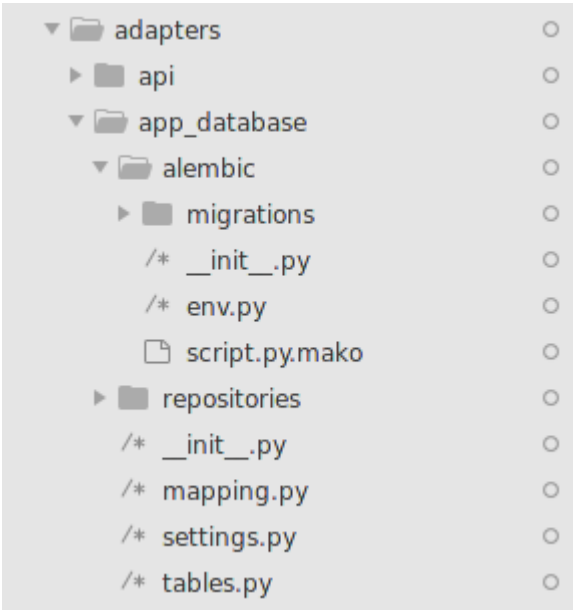
Слой адаптеров

В адаптерах лежат интеграции со внешними сервисами. Там же лежат и буквы VC из MVC веб библиотеки, CLI, продьюсеры, консьюмеры и прочие интеграционные компоненты (например api клиенты).

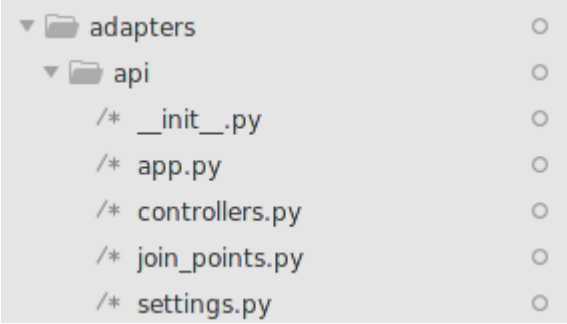


Работа с базой данных (описание таблиц, миграции, код запросов) лежат во вторичных адаптерах. Нужно понимать, что в компании есть разные СУБД (преимущественно MSSQL и Postgres), **у нас используется как можно меньше диалекто-зависимых конструкций, а если используется, то помечается как # TODO: dialect dependent**

- таблицы описаны в виде обычных таблиц sqlalchemy, **указывать naming_convention в Metadata обязательно**. Если планируется использовать MSSQL, то указывается схема "app", так же в env.py алембика делаются изменения (см. в демо проекте).



- названия таблиц начинаются с маленьких букв, используется snake_case. Таблицы-сущности и справочники именуются во множественном числе, всяческие таблицы-логи в единственном числе
- если в проекте сложная доменная логика, то создается маппер таблиц на классы с бизнес-логикой через императивный маппинг sqlalchemy (тут важно следить за связанными сущностями для избегания проблемы N+1)
- весь код запросов пишется в репозиториях. Бизнес логика не зашивается в запросы и слой адаптеров
- в сложном проекте из репозиториев возвращаются намапленные ORM объекты, в простом случае – DTO (data transfer object)
- код для создания веб приложения, его настройка, регистрирование контроллеров, расположено в адаптерах

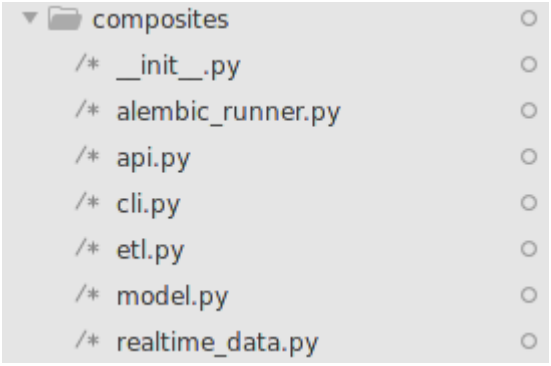


- в контроллеры внедряются сервисы из слоя приложения, контроллеры вызывают методы сервисов
  - именно тут происходит подготовка структур из слоя бизнес логики для передачи наружу. Самое частое - сформировать некую структуру и подготовить ее для сериализации
- похожая история происходит с консьюмерами асинхронных задач

Сериализация:

- Decimal - строка
- datetime - isoformat (см. "Работа с датой (timezone)")
- time - isoformat
- UUID - строка
- Enum - тут либо name, либо value, в зависимости от потребностей

Слой композитов



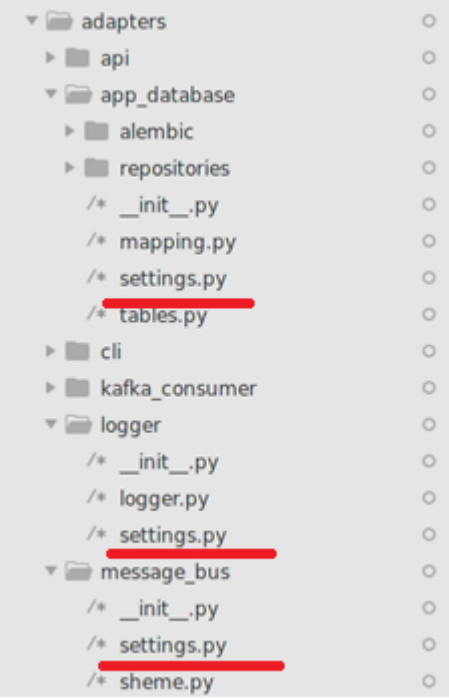
Тут происходит сборка компонентов для запуска процесса. Именно тут инициализируются настройки, внедряются зависимости.

Например, для сборки HTTP API нужно:

- инстанцировать engine sqlalchemy
- создать connection
- внедрить его в репозитории
- инстанцировать другие компоненты со внедренными зависимостями
- инстанцировать сервисы
- передать сервисы в фабрику http app (слой адаптеров)
- фабрика зарегистрирует контроллеры с внедренными сервисами
- для запуска нужно натравить gunicorn на объект app(Falcon) в модуле композита

Настройки проекта

Все настройки передаются в виде env параметров. Создаются settings.py в каждой заинтересованной части приложения (для СУБД приложения будет свой settings.py в своем адаптере, для http api свои настройки). В settings.py создается класс-наследник BaseSettings от pydantic.



Композит инстанцирует каждый класс настроек и **внедряет** в нужные компоненты слоев кода

Как что взаимодействует

Первичный адаптер (такой как контроллер или консьюмер) вызывает сервис из слоя приложения, передает туда параметры, сервис валидирует параметры. Для выполнения бизнес-логики могут потребоваться данные из интеграций, в этом случае сервис через интерфейс вызовет логику адаптера и получит данные. Нотификация, публикация сообщений тоже находятся в адаптерах и вызов так же будет через интерфейсы.

Главное понимать, что точка входа в слой бизнес логики – сервис. Слой бизнес логики не зависит от интеграций, то есть слой приложения не должен нигде импортировать что-то из адаптеров, композитов.

Обработка ошибок

Клиенты не должны знать технические детали ошибок.
Для этого все ошибки приложения описаны в слое бизнес логики. Так же есть валидация в сервисах. Ошибки из СУБД мы не используем, они слишком обобщенные. Для этого сервис может проверить некоторые условия и возбудить ошибку из своего слоя.
В слое адаптеров происходит отлов ошибок из слоя бизнес логики. Далее ошибка трансформируется в нужную структуру для адаптера. Например, все ошибки слоя бизнес логики можно унаследовать от одного типа и отловить его на уровне web библиотеки для трансформации.

Работа с датой (timezone)

В БД приложения все время хранится в UTC, на фронт тоже уходит UTC, на беке расчеты тоже происходят по UTC, фронт уже сам решит что с этим делать.

Даты бывают aware – с указанием тайм-зоны, и naive - без указания

- 2021-08-30T8:00:0.00000+07:00 - 8 утра по Новокузнецку GMT+7
- 2021-08-30T1:00:0.00000+00:00 – все те же 8 утра по Новокузнецку GMT+7, но записанные в тайм-зоне UTC
- 2021-08-30T8:00:0.00000 – 8 утра но не понятно где.

У нас БД MS-SQL, принято решение, что все даты мы должны хранить в naive utc виде, в поле типа datetime (нет возможности хранения тайм-зоны), т.е. 8 утра Новокузнецка в нашей базе будет храниться как 2021-08-30T1:00:0.00000 (зона UTC, naive), на фронт дата должна уходить с указанием тайм зоны 2021-08-30T1:00:0.00000+00:00.

Соответственно если есть задача написать ETL, то надо узнавать в какой таймзоне хранятся даты и преобразовывать. Так же при запуске фоновых задач, бывает нужно указать таймзону, такое выносится в настройки проекта и прокидывается в сервис при сборке в композите.

Логирование

print не используется в основных ветках проекта!
Используется стандартный модуль logging. Настройки хранятся в каждом settings.py, далее композит собирает итоговый конфиг и происходит настройка (код по слиянию конфигов можно держать в адаптере). Используется такой формат:

```
fmt = '%(asctime)s.%(msecs)03d [%(levelname)s]|[%(name)s]: %(message)s'
datefmt = '%Y-%m-%d %H:%M:%S'
```

Уровни используются стандартные. **Глобальные объекты логгера не используются.**
Хочется обратить внимание на частую ошибку использования:
Неверно

```
id_ = 1
self.logger.info(f'Transport with id [{id_}] was deleted')
```

Верно

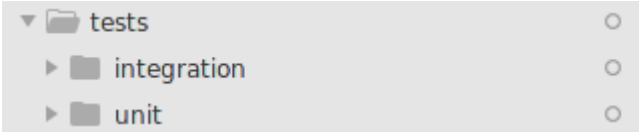
```
id_ = 1
self.logger.info('Transport with id [%s] was deleted', id_)
```

Так при многочисленных DEBUG логах на ландшафте с уровнем INFO мы не скушаем кучу процессорного времени на форматирование строк. Наша инфраструктура собирает логи, для этого нужно их писать в JSON формате. Для этого используется пакет **python-json-logger**. JSON формат в наших проектах опционален, настраивается через settings.py.

Отчеты и аналитика

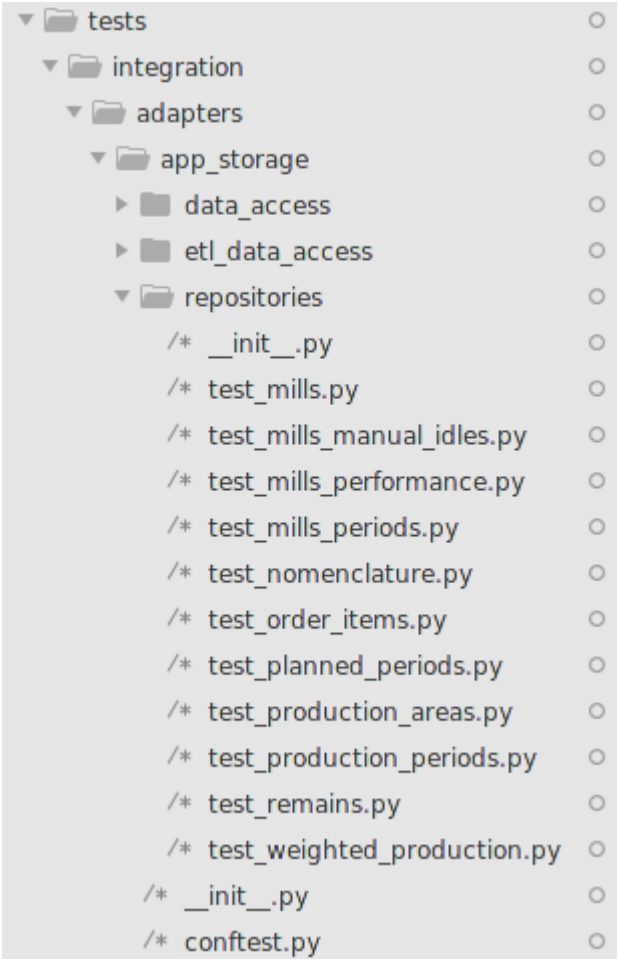
Часто в проектах требуется аналитика и отчеты. Тут не имеем ничего против тяжелых sql запросов, но хотелось бы поменьше диалекто-зависимых конструкций. Если ответ от СУБД обрабатывать не надо, то репозиторий можно внедрить напрямую в контроллер отдавать результат минуя сервисы.

Тесты



В проектах пишутся юнит тесты и интеграционные. Юниты в приоритете и их обычно больше. В юнит тестах адаптеры мокаются. Из-за инфраструктурных особенностей интеграционные тесты репозиториев используют sqlite в ОЗУ.

В каталогах нужно отразить структуру проекта. Имена файлам давать по именам модулей или классов. В именах тестах отражать кейс, например test__<class_name>__<case>.



Прочие технические детали

Транзакции СУБД

У нас нет вложенных транзакций. У нас используется что-то типа паттерна "Единица работы". "Единица работы" в наших внутренних библиотеках работает как декоратор над методом контроллера И/ИЛИ над публичным методом сервиса. "Единица работы" открывает транзакцию перед выполнением основного кода, закрывает после, отслеживает попытки создать внутренние транзакции и мешает этому. В коде не должно быть ручного управления транзакциями.

Асинхронный код

Чтобы использовать асинхронные решения нужно это обосновать. У нас очень мало нагруженных приложений, которые действительно упираются в IO. Если очень нужно, то используем gevent, просто патчим код в начале запуска. Отдельно надо патчить mssql драйвер.

Зависимости DS

Такие библиотеки как pandas, numpy используем в DS модулях (feature engineering и тд.), **вне этих модулей не используется**.

Websockets

Вебсокеты строятся **при необходимости получать на фронте push уведомления**. Используется Rabbitmq с плагином Stomp over ws, тык .

Авторизация

Наш фронт ходит в keycloak и получает токен, этот токен является JWT токеном и расшифровывается стандартными средствами, мы используем пакет pyjwt. В токене находится вся информация о пользователе. Токен передается как bearer в стандартном заголовке Authorization (пример: Authorization: Bearer ...). Приложение принимает решение на основе данных из токена. У нас есть

класс "Аутентификатор", который этим занимается, он внедряет в контроллер инфу о пользователе. Контроллер принимает решения об авторизации.

Мониторинг

Тут пока пусто, инфраструктура пока еще не готова, нет централизованных решений

Оформление кода

Код пишется по PEP8, докстринги по PEP256, 257.

Есть конфиг для yapf и isort, строки переносим при достижении 80 символов. В крайнем случае можно переносить на 100 (тут надо отключать автоформаттер).
Сначала запускается isort, потом yapf.

Необходимо следить за размером модулей, классов, методов. При распухании происходит декомпозиция и рефакторинг.