точка для предпринимателей и предприятий

Опыт использования FastAPI совместно с Clean Architecture

Преимущества и ограничения

Виктор Луферов

Кандидат технических наук

Более 12 лет в IT

Более 6 лет опыта работы с Python

Tech Lead в Точке, ранее — Tech Lead в Сбере

точка

План на сегодня

- Clean Architecture

точка

План на сегодня

- Clean Architecture
- Dependency Injection в FastAPI

точка

План на сегодня

- Clean Architecture
- Dependency Injection в FastAPI
- Управление транзакциями

точка

План на сегодня

- Clean Architecture
- Dependency Injection в FastAPI
- Управление транзакциями
- Обработка ошибок

«Архитектура —
это о стоимости изменений.
Хорошая архитектура делает
изменения дешевыми.
Плохая — дорогими».

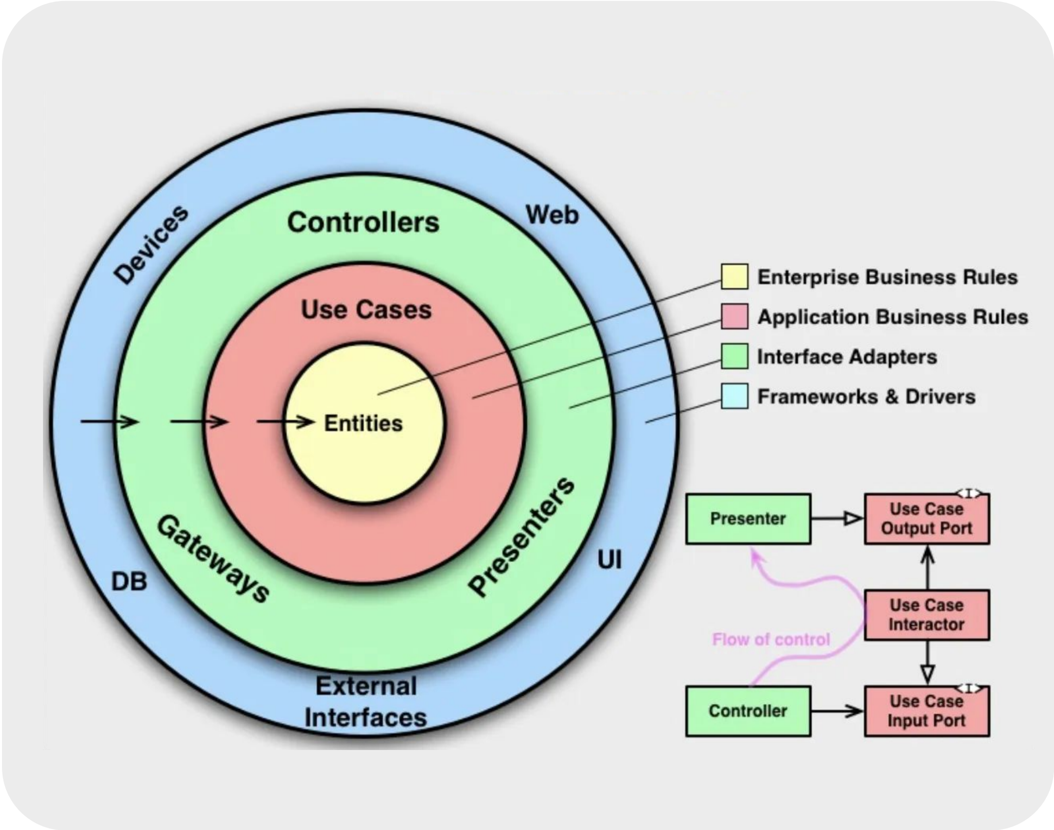
© Дядя Боб

точка

Требования

- Независимость от фреймворка
- Масштабируемость
- Сложная бизнес-логика
- Высокие требования тестируемости

Clean Architecture



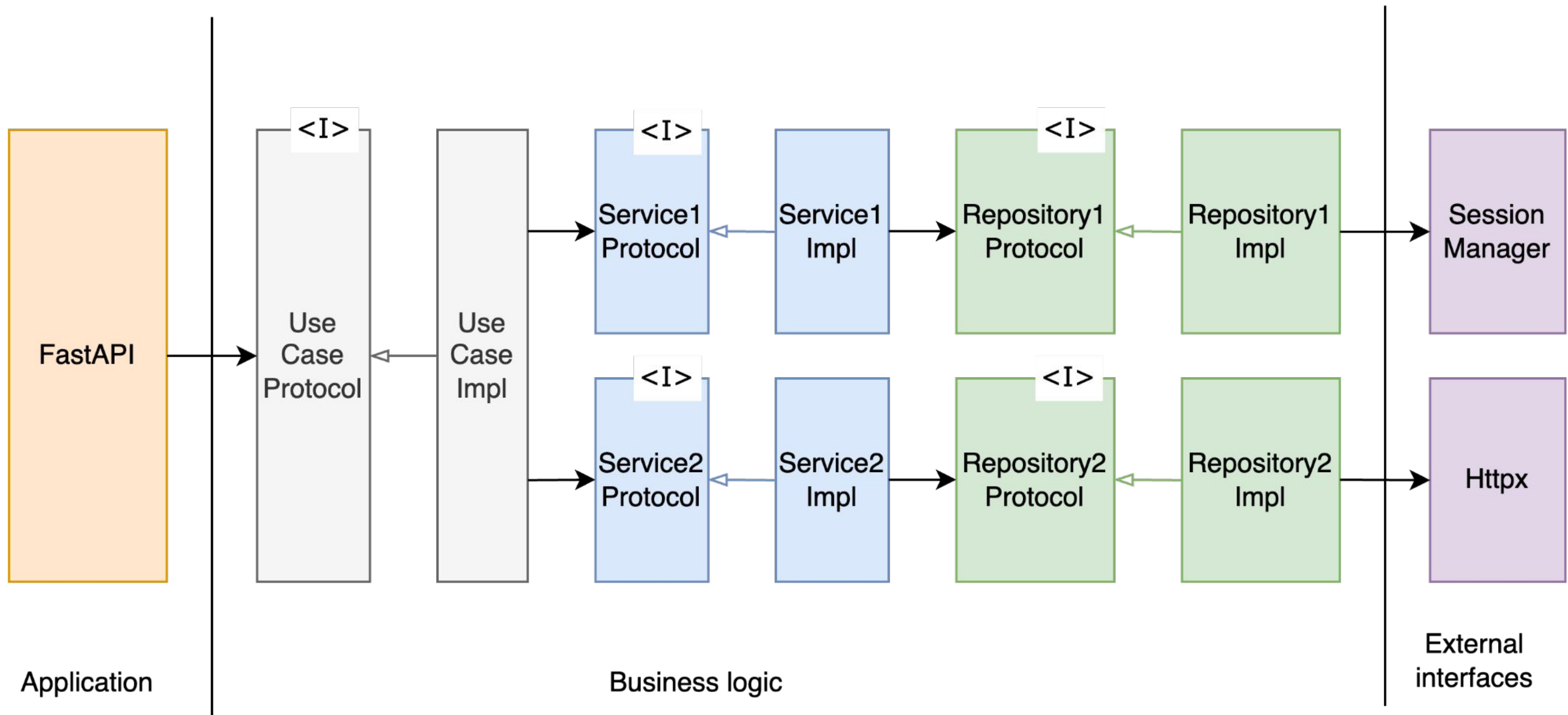
точка

SOLID в помощь

- S — Single Responsibility Principle
- O — Open-Closed Principle
- L — Liskov Substitution Principle
- I — Interface Segregation Principle
- D — Dependency Inversion Principle

Clean Architecture

точка

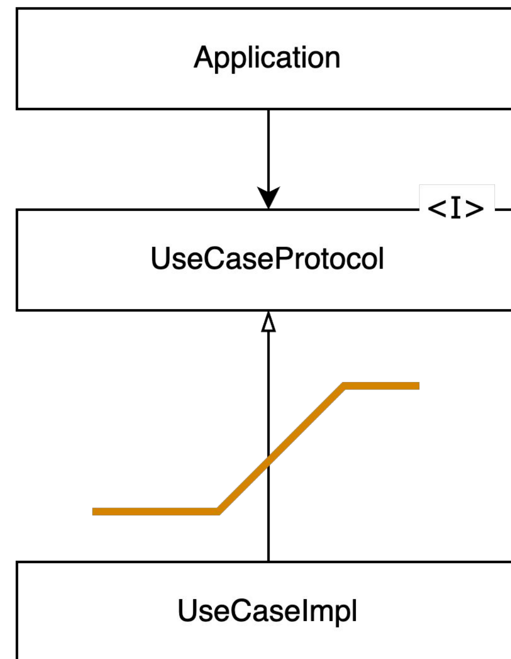


D – Dependency Inversion

точка

Protocol vs ABC

Критерий	Protocol (PEP 544)	ABC
Гибкость	✅ Лучше	❌
Проверка в runtime	❌ но можно @runtime_checkable	✅ Лучше
Множественные интерфейсы	✅ Лучше	❌
Явное наследование	❌	✅ Лучше

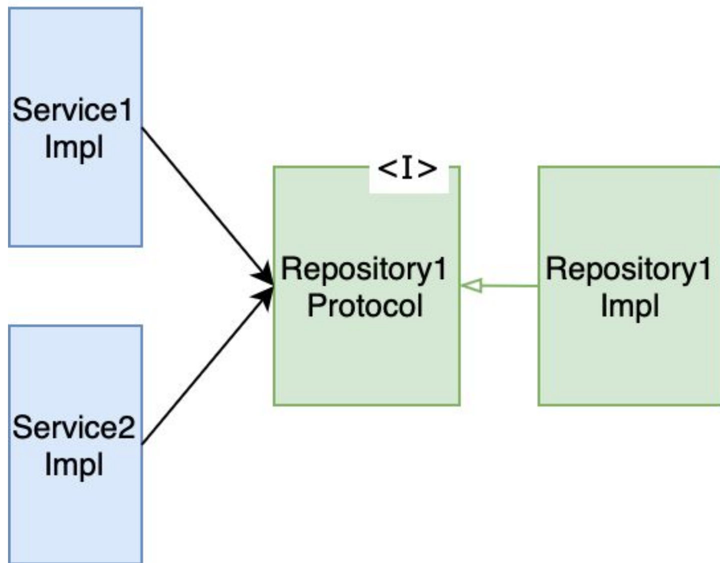


В каком месте писать интерфейсы

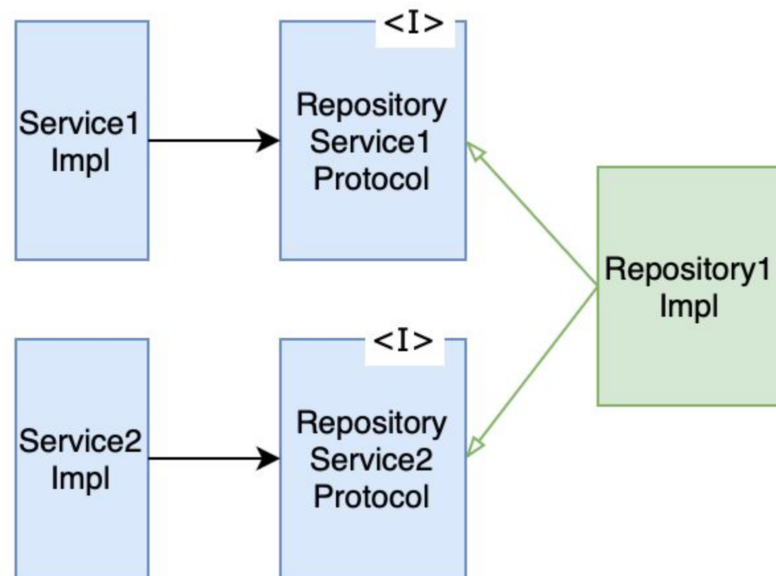
точка

- Чтобы соблюдать принцип I из SOLID, нужно писать на принимающей
- Мы пишем на отдающей

на отдающей стороне

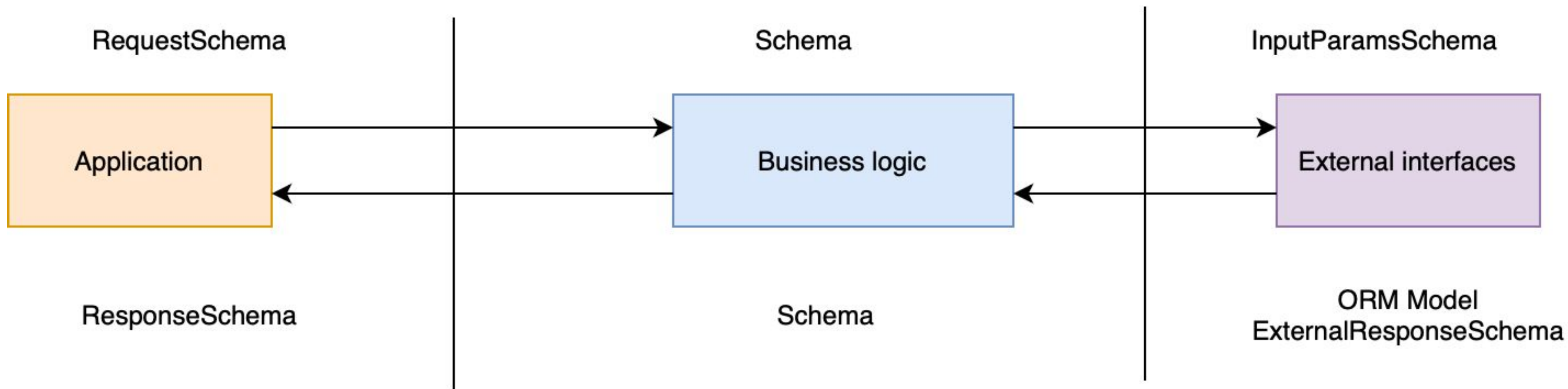


на принимающей



Изоляция слоев

точка



Наши правила

- 1 роут = 1 UseCase
- UseCase ← Сервисы ← Репозитории
- Общение между слоями через DTO
- Анемичная модель

точка

Dependency Injection в FastAPI

Хотелось бы DI & IoC-контейнер

- Представляет собой реестр (контейнер) объектов, которыми он управляет
- Позволяет декларативно конфигурировать объекты и их свойства
- Код классов не должен зависеть от IoC-фреймворка
- Берёт на себя:

✔ Управление жизненным циклом объектов: создание, удаление
✔ Управление зависимостями между объектами

А как же другие DI?

FastAPI Depends:

- ⭐ Github stars: 84.3k
- Нативная, только FastAPI
- Scope: RequestScope, Singleton через lru_cache
- Синхронный и асинхронный

Dishka:

- ⭐ Github stars: 656
- Не привязана к фреймворку
- Scope: Request, Application, Session
- Синхронный и асинхронный
- Один из самых популярных провайдеров, который поддерживает асинхронные фабрики

Сервис регистрации пользователей

точка

```
1.  from typing import Protocol
2.  from app.schemas.auth import UserCreate, UserOut, Token
3.
4.  class AuthServiceProtocol(Protocol):
5.      async def register_user(self, user_data: UserCreate) -> UserOut:
6.          ...
7.
8.      async def get_current_user(self, token: str) -> UserOut | None:
9.          ...
10.
11. class AuthServiceImpl:
12.     def __init__(self, users_repository: UsersRepository) -> None:
13.         self.users_repository = users_repository
14.
15.     async def register_user(self, user_data: UserCreate) -> UserOut:
16.         return await self.users_repository.create(user_data)
```

Dishka. Создаём контейнер

точка

```
1.  from dishka import Provider, Scope, provide, make_async_container
2.  from app.services.auth import AuthServiceImpl, AuthService
3.  from app.repositories.users import UserRepositoryImpl, UserRepository
4.
5.  class AuthProvider(Provider):
6.      scope = Scope.REQUEST
7.
8.      @provide
9.      def provide_user_repository(self) -> UserRepository:
10.         return UserRepositoryImpl()
11.
12.     @provide
13.     def provide_auth_service(self, user_repository: UserRepository) -> AuthService:
14.         return AuthServiceImpl(user_repository)
15.
16. providers = [AuthProvider()]
17. container = make_async_container(*providers)
```

Dishka. Запускаем FastAPI приложение

точка

```python
1.  from fastapi import FastAPI
2.  from dishka.integrations.fastapi import (
3.      DishkaRoute,
4.      FromDishka,
5.      inject,
6.      setup_dishka,
7.  )
8.
9.  app = FastAPI()
10.
11. app.router.route_class = DishkaRoute    # <- или @inject
12. setup_dishka(container, app)
13.
14. @app.post("/register", response_model=UserOut)
15. @inject         # <- если не app.router.route_class = DishkaRoute
16. async def register(user_data: UserCreate, auth_service: FromDishka[AuthService]):
17.     return await auth_service.register_user(user_data)
```

Я бы советовал использовать **Dishka** в проектах

FastAPI + Dishka. get_current_user. У вас теперь два провайдера

точка

```
1.   def get_current_user(
2.       auth_service: FromDishka[AuthService],
3.       token: Annotated[str, Depends(lambda: 'token')]
4.   ) -> UserOut:
5.
6.
7.   @app.post("/register", response_model=UserOut)
8.   @inject          # <- если не app.router.route_class = DishkaRoute
9.   async def register(
10.      user_data: UserCreate,
11.      auth_service: FromDishka[AuthService],
12.      current_user: Annotated[UserOut, Depends(get_current_user)],
13.  ):
14.      if not current_user:
15.          raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED)
16.      return await auth_service.register_user(user_data)
```

FastAPI. Описываем провайдер

точка

Файлы depends.py — описание контейнера

- agent
  - apps
    - ia
    - integrations
    - scheduler
      - repositories
      - services
      - use_cases
      - __init__.py
      - depends.py
      - enums.py
      - models.py
      - router.py
      - schemas.py
      - tasks.py

```python
# depends.py
from typing import Annotated
from fastapi import Depends

from fast_clean.depends import SessionManager

from .repositories import UsersRepositoryProtocol, UsersRepositoryImpl
from .services import AuthServiceProtocol, AuthServiceImpl
from .use_cases import UserRegisterProtocol, UserRegisterImpl

# --- repositories ---
def get_users_repository(session_manager: SessionManager) -> UsersRepositoryProtocol:
    return UsersRepositoryImpl(session_manager)

UserRepository = Annotated[UsersRepositoryProtocol, Depends(get_users_repository)]

# --- services ---
def get_auth_services(user_repository: UserRepository) -> AuthServiceProtocol:
    return AuthServiceImpl(user_repository)

AuthService = Annotated[AuthServiceProtocol, Depends(get_auth_services)]

# --- use cases ---
def get_user_register_use_case(auth_service: AuthService, user_data: UserCreate) ->
UserRegisterProtocol:
    return UserRegisterImpl(auth_service)

UserRegisterUseCase = Annotated[UserRegisterProtocol, Depends(get_user_register_use_case)]
```

FastAPI. Запускаем приложение

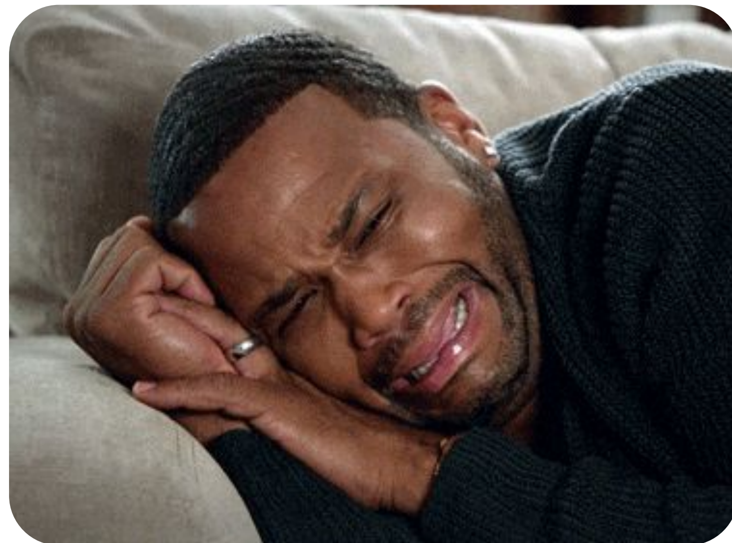
точка

```python
1.  from fastapi import FastAPI
2.
3.  app = FastAPI()
4.
5.  @app.post("/register")
6.  async def register(user_register_use_case: UserRegisterUseCase):
7.      return await user_register_use_case()
```

Но есть проблема

Нативный Depends из **FastAPI** нигде больше не работает,

за исключением **FastStream** 🚀



«Внедряй руками, в чём проблема?»

© Любой Golang-разработчик

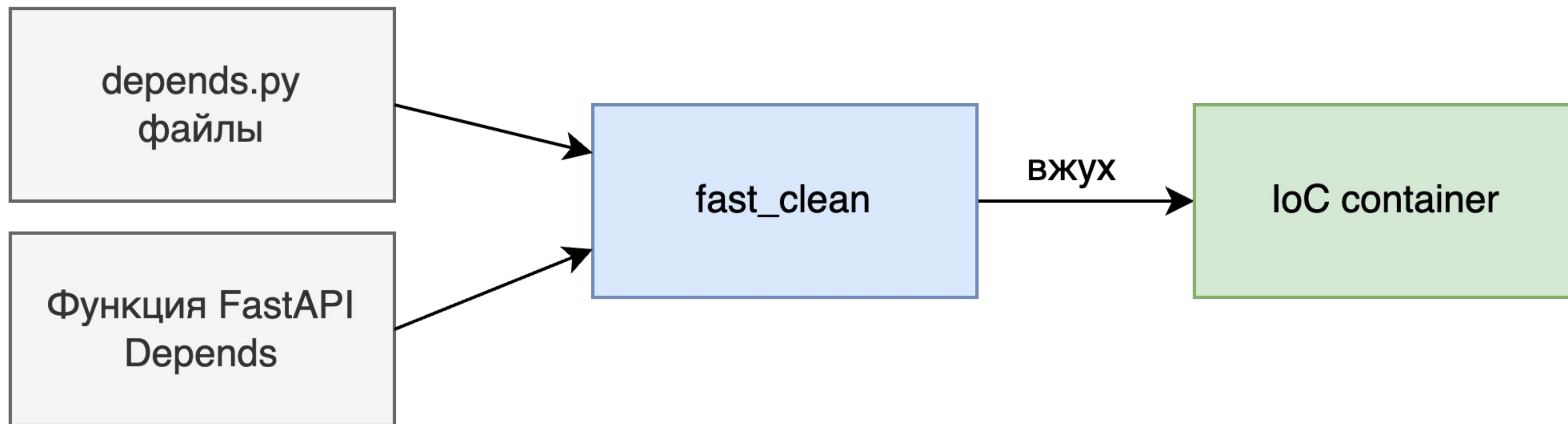
Собираем руками

точка

```
@flow
async def pull_flow() -> None:
    """
    Поток, обрабатывающий индексы типа pull.
    """
    async with asynccontextmanager(get_flow_async_session)()  as session, make_pull_index_service(
        session
    ) as pull_index_service:
        index_offset_repository_factory =  await get_index_offset_repository_factory()
        index_repository =  await get_index_repository(session)
        index_iterator_service =  await
get_index_iterator_service(index_offset_repository_factory, index_repository)
        flow_settings_repository =  await get_flow_settings_repository()
        handle_pull_indices_chunk_use_case =  await get_handle_pull_indices_chunk_use_case(
            pull_index_service,
            index_iterator_service,
            index_repository,
            flow_settings_repository,
        )
        await handle_pull_indices_chunk_use_case()
```

Делаем свой IoC контейнер

точка



https://github.com/Luferov/fast-clean

FastAPI. Нативный контейнер

точка

```
1.  from fast_clean.depends import get_container
2.
3.  from app.apps.depends import UserRegisterUseCaseProtocol
4.
5.  async def execution_task(user_data: UserCreate) -> None:
6.      async with get_container() as container:
7.          user_register_use_case = await
    container.get_by_type(UserRegisterUseCaseProtocol)
8.          await user_register_use_case(user_data)
```

Теперь можно использовать везде, включая **Prefect**, **APScheduler** и **REPL**

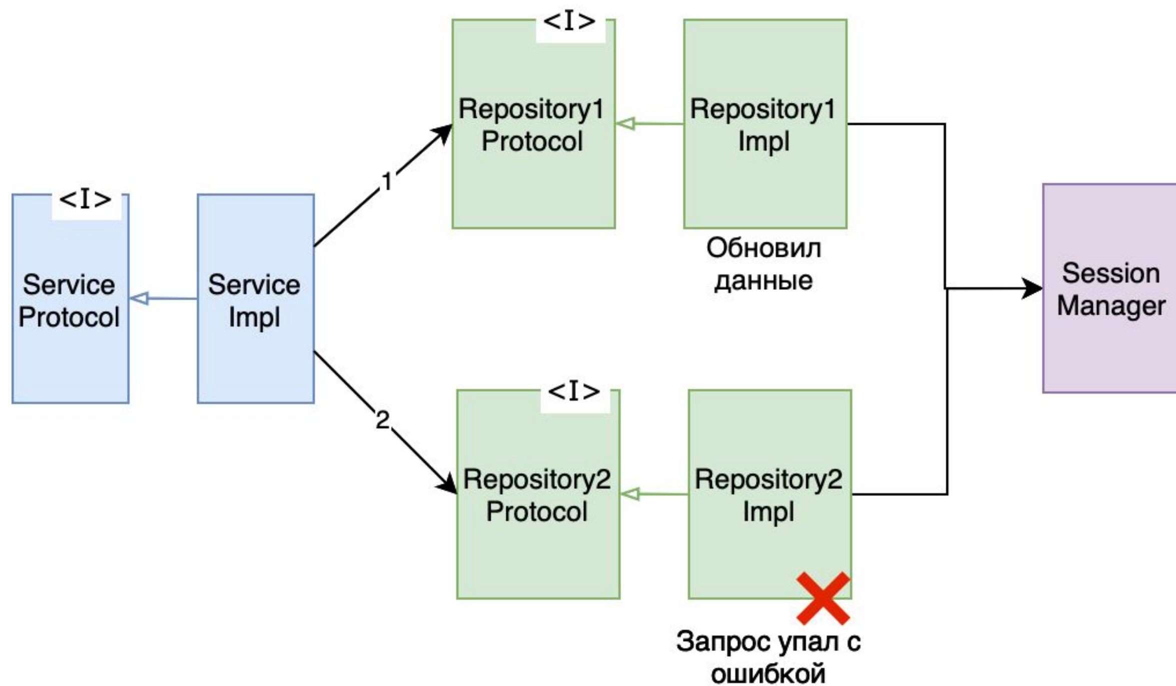
точка

Управление транзакциями

Проблема атомарности

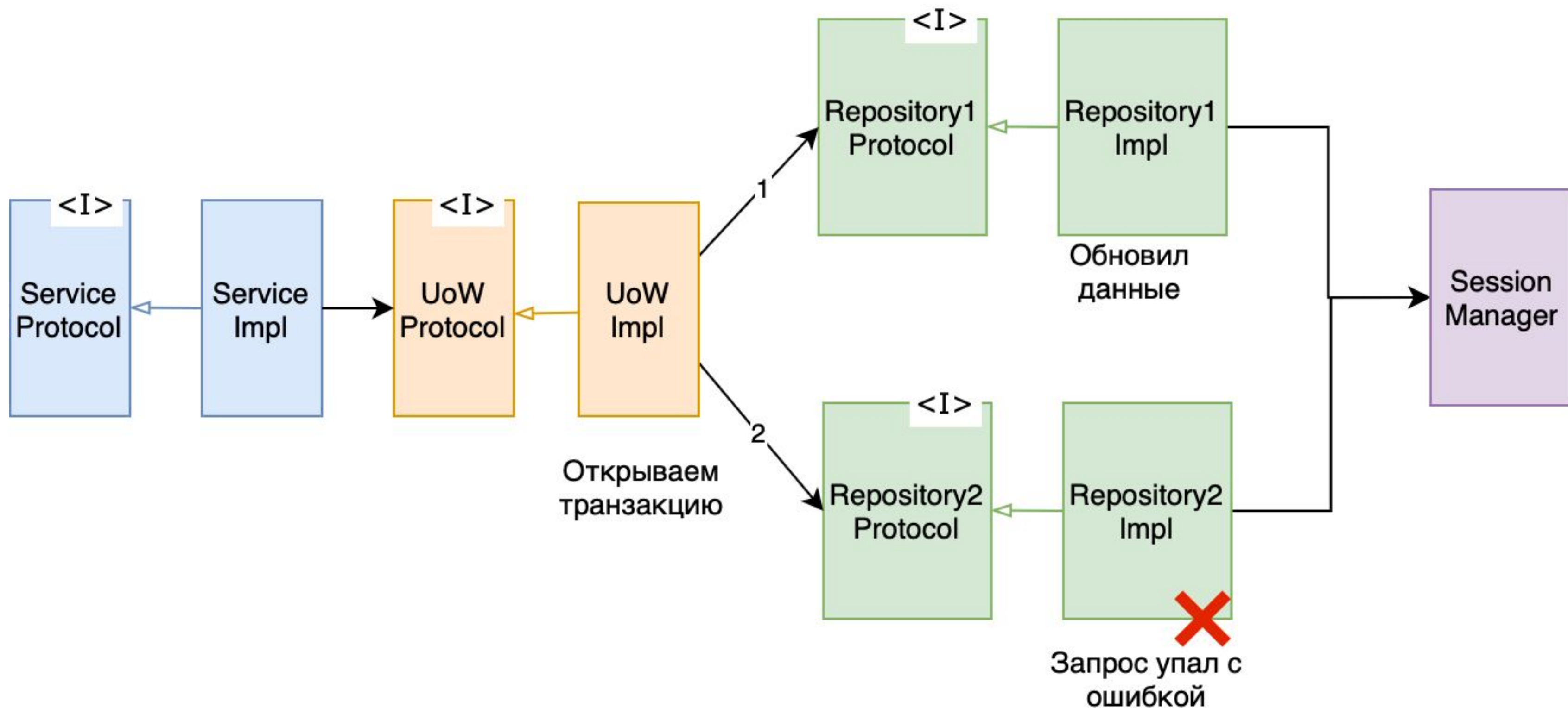
точка

Если в Repository 1 запрос выполнился, а в Repository 2 упал с ошибкой, то в Repository 1 отката не будет



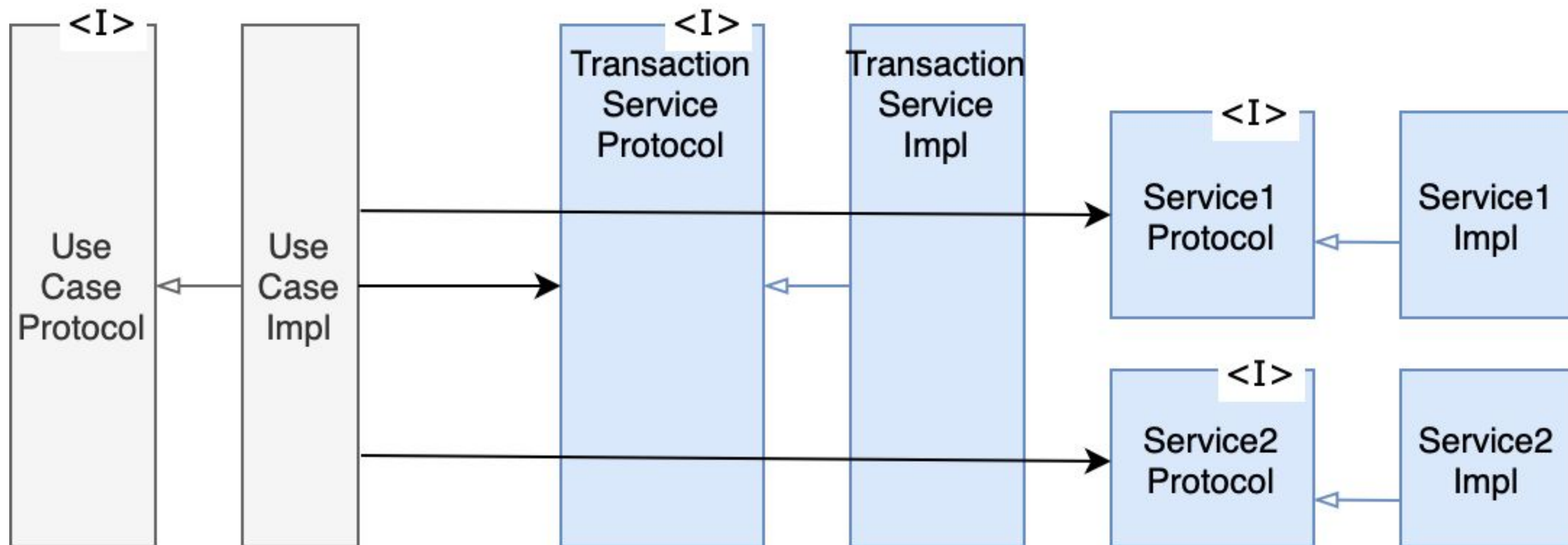
Вспоминаем про UnitOfWork

точка



Добавляем TransactionService

точка



Добавляем TransactionService

точка

```python
from contextlib import asynccontextmanager
from typing import AsyncContextManager, AsyncIterator, Protocol, Self
import sqlalchemy as sa
from sqlalchemy.ext.asyncio import AsyncSession

class TransactionServiceProtocol(Protocol):
    def begin(self: Self, immediate: bool = True) -> AsyncContextManager[None]:
        ...

class TransactionServiceImpl:
    def __init__(self, session: AsyncSession) -> None:
        self.session = session

    @asynccontextmanager
    async def begin(self: Self, immediate: bool = True) -> AsyncIterator[None]:
        async with self.session.begin():
            if immediate:
                await self.session.execute(sa.text('SET CONSTRAINTS ALL IMMEDIATE'))
            yield
```

DEFERRED — откладывает проверку ограничений до коммита
IMMEDIATE — принудительно проверяет все ограничения

Добавляем SessionManager

точка

Добавляем SessionManager — прослойка между Repository и AsyncSession

```
1.  class SessionManagerProtocol (Protocol):
2.      def get_session(self: Self, immediate: bool = True ) -> AsyncContextManager[AsyncSession]:
3.          ...
4.
5.  class SessionManagerImpl :
6.      def __init__(self, session: AsyncSession ) -> None:
7.          self.session = session
8.
9.      @asynccontextmanager
10.     async def get_session(self: Self, immediate: bool = True ) -> AsyncIterator[AsyncSession]:
11.         if self.session.in_transaction():
12.             async with self.session.begin_nested():
13.                 yield self.session
14.         else:
15.             async with self.session.begin():
16.                 if immediate:
17.                     await self.session.execute(sa.text( 'SET CONSTRAINTS ALL IMMEDIATE' ))
18.                 yield self.session
```

Описываем зависимости

точка

```
1.  # depends.py
2.  async def get_async_session(settings_repository: SettingsRepository) ->
    AsyncIterator[AsyncSession]:
3.      """Получаем асинхронную сессию."""
4.      async with SessionFactory.make_async_session_static(settings_repository) as session:
5.          yield session
6.
7.  Session = Annotated[AsyncSession, Depends(get_async_session)]
8.
9.  def get_session_manager(session: Session) -> SessionManagerProtocol:
10.     """Получаем менеджер сессий."""
11.     return SessionManagerImpl(session)
12.
13. SessionManager = Annotated[SessionManagerProtocol, Depends(get_session_manager)]
```

Использование в UseCase

точка

```
1.  from typing import Self
2.  from fast_clean.use_cases import UseCaseProtocol
3.  from fast_clean.depens import TransactionServiceProtocol
4.
5.  from ..schemas import UserRegisterResponseSchema
6.  from ..services import Service1Protocol, Service2Protocol
7.
8.  UserRegisterUseCaseProtocol = UseCaseProtocol[UserRegisterResponseSchema]
9.
10. class UserRegisterUseCaseImpl:
11.    def __init__(self,
12.        transaction_service: UserRegisterResponseSchema,
13.        service1: Service1Protocol,
14.        service2: Service2Protocol,
15.    ) -> None:
16.        self.transaction_service = transaction_service
17.        self.service1 = service1
18.        self.service2 = service2
19.
20.    async def __call__(self: Self) -> UserRegisterResponseSchema:
21.        async with self.transaction_service.begin():
22.            await self.service1.do_smth()
23.            return await self.service2.do_smth()
```

Использование в Repository

точка

```
1.  class UserRepositoryProtocol(
2.      CrudRepositoryProtocol[UserReadSchema, UserCreateSchema, UserUpdateSchema],
3.      Protocol,
4.  ):
5.
6.      async def get_by_id(
7.          self: Self, user_id: uuid.UUID, *, with_profile:bool = False
8.      ) -> UserReadSchema | None:
9.          ...
10.
11. class UserRepositoryImpl(
12.     DbCrudRepository[User, UserReadSchema, UserCreateSchema, UserUpdateSchema]
13. ):
14.
15.     async def get_by_id(
16.         self: Self, user_id: uuid.UUID, *, with_profile:bool = False
17.     ) -> UserReadSchema | None:
18.         async with self.session_manager.get_session() as s:
19.             statement = sa.select(self.model_type).where(self.model_type.id == integration_id)
20.             if with_profile:
21.                 statement = statement.options(selectinload(self.model_type.profile))
22.             model = (await s.execute(statement)).scalar_one_or_none()
23.             return self.model_validate(model) if model is not None else None
```

точка

Обработка ошибок

точка

Наименование Exception vs Error

Exception — непредвиденные ошибки, которые возникают в результате работы программы при неконтролируемых действиях

Error — ошибки бизнес-логики, пользовательские ошибки

точка

Наименование Exception vs Error

What it does

Checks for custom exception definitions that omit the `Error` suffix.

Why is this bad?

The `Error` suffix is recommended by PEP 8:

> Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix `"Error"` on your exception names (if the exception actually is an error).

Example

```
class Validation(Exception): ...
```

Use instead:

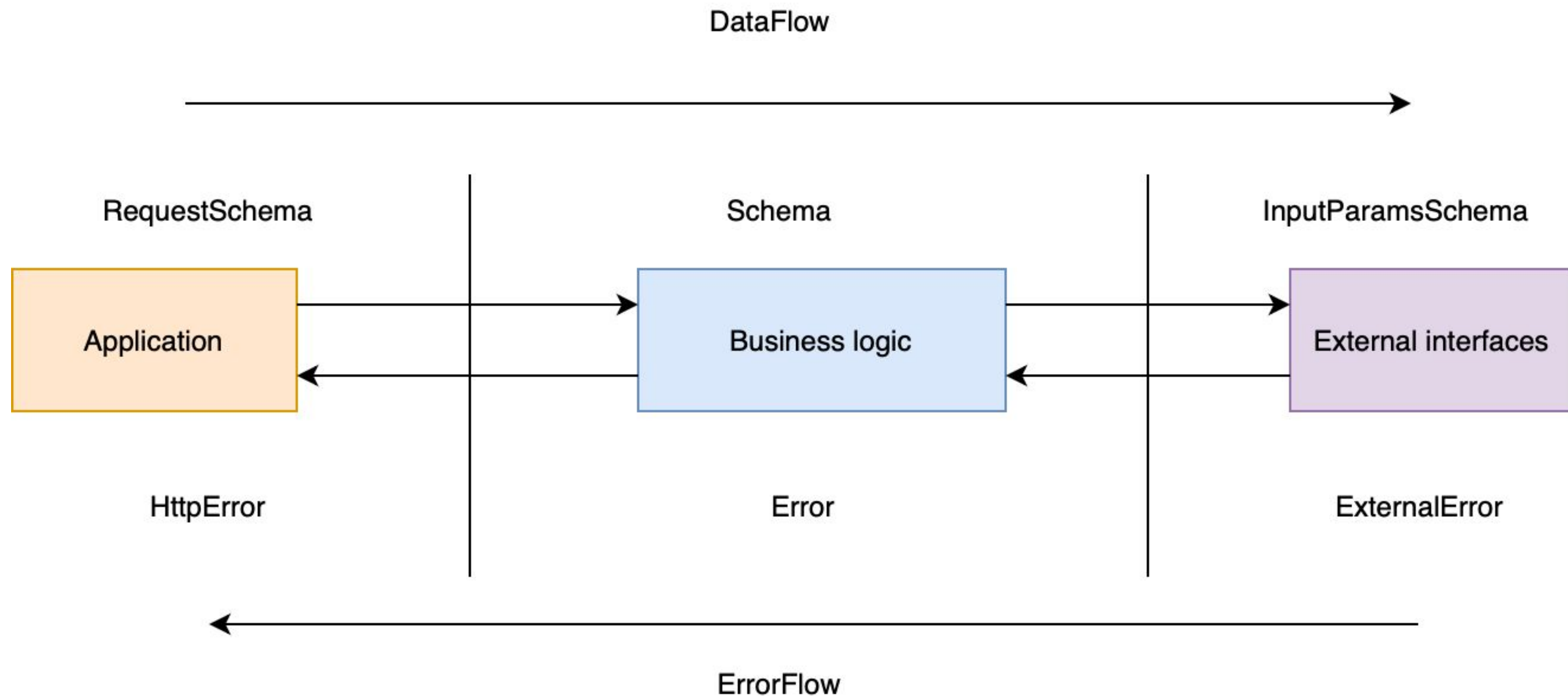
```
class ValidationError(Exception): ...
```

https://docs.astral.sh/ruff/rules/error-suffix-on-exception-name/

PEP8 говорит:
Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix “Error” on your exception names (if the exception actually is an error)

Какие у нас есть ошибки

точка



точка

Ограничения

- Изолируй ошибки библиотек от бизнес-логики
- Преобразуй ошибки бизнес-логики в HTTP-ответы

Описываем самое базовое исключение проекта

точка

```
1.  class BusinessLogicException(Exception, ABC):
2.      """Базовое исключение бизнес-логики."""
3.  
4.      @property
5.      def type(self: Self) -> str:
6.          """Тип ошибки."""
7.          return snakecase(type(self).__name__.replace('Error', ''))
8.  
9.      @property
10.     @abstractmethod
11.     def msg(self: Self) -> str:
12.         """Сообщение ошибки."""
13.         ...
14. 
15.     def __str__(self: Self) -> str:
16.         return self.msg
17. 
18.     def get_schema(self: Self, debug: bool) -> BusinessLogicExceptionSchema:
19.         return BusinessLogicExceptionSchema(
20.             type=self.type,
21.             msg=self.msg,
22.             traceback=(''.join(traceback.format_exception(type(self), self, self.__traceback__)) if debug else
    None),
23.         )
```

Определяем формат сообщения для фронта

Описываем возможные ошибки проекта

точка

```
1.  class PermissionDeniedError (BusinessLogicException ):
2.      """Ошибка, возникающая при недостатке прав для выполнения действия."""
3.
4.      @property
5.      def msg(self: Self) -> str:
6.          return 'Недостаточно прав для выполнения действия'
7.
8.  class ModelIntegrityError (BusinessLogicException ):
9.      """Ошибка целостности данных при взаимодействии с моделью."""
10.
11.     def __init__(self, model: type[ModelType] | str, action: ModelActionEnum, *args: object  ) -> None:
12.         super().__init__(*args)
13.         self.model = model
14.         self.action = action
15.
16.     @property
17.     def msg(self: Self) -> str:
18.         """Сообщение ошибки."""
19.         msg = 'Ошибка целостности данных'
20.         model_name = self.model  if isinstance(self.model, str) else self.model.__name__
21.         match self.action:
22.             case ModelActionEnum.INSERT:
23.                 msg += f' при создании модели {model_name}'
24.             case ModelActionEnum.UPDATE:
25.                 msg += f' при изменении модели {model_name}'
26.             case ModelActionEnum.UPSERT:
27.                 msg += f' при создании или изменении модели {model_name}'
28.             case ModelActionEnum.DELETE:
29.                 msg += f' при удалении модели {model_name}'
30.         return msg
```

Описываем handlers для ошибок

точка

```
1.  async def business_logic_exception_handler(
2.      settings: CoreSettingsSchema, request: Request, exception: BusinessLogicException
3.  ) -> Response:
4.      """Обработчик базового исключения бизнес-логики."""
5.      return await http_exception_handler(
6.          request,
7.          HTTPException(
8.              status_code=status.HTTP_400_BAD_REQUEST,
9.              detail=[exception.get_schema(settings.debug).model_dump()],
10.         ),
11.     )
12.
13.
14. async def permission_denied_error_handler(
15.     settings: CoreSettingsSchema, request: Request, error: PermissionDeniedError
16. ) -> Response:
17.     """Обработчик ошибки, возникающей при недостатке прав для выполнения действия."""
18.     return await http_exception_handler(
19.         request,
20.         HTTPException(
21.             status_code=status.HTTP_403_FORBIDDEN,
22.             detail=[error.get_schema(settings.debug).model_dump()],
23.         ),
24.     )
```

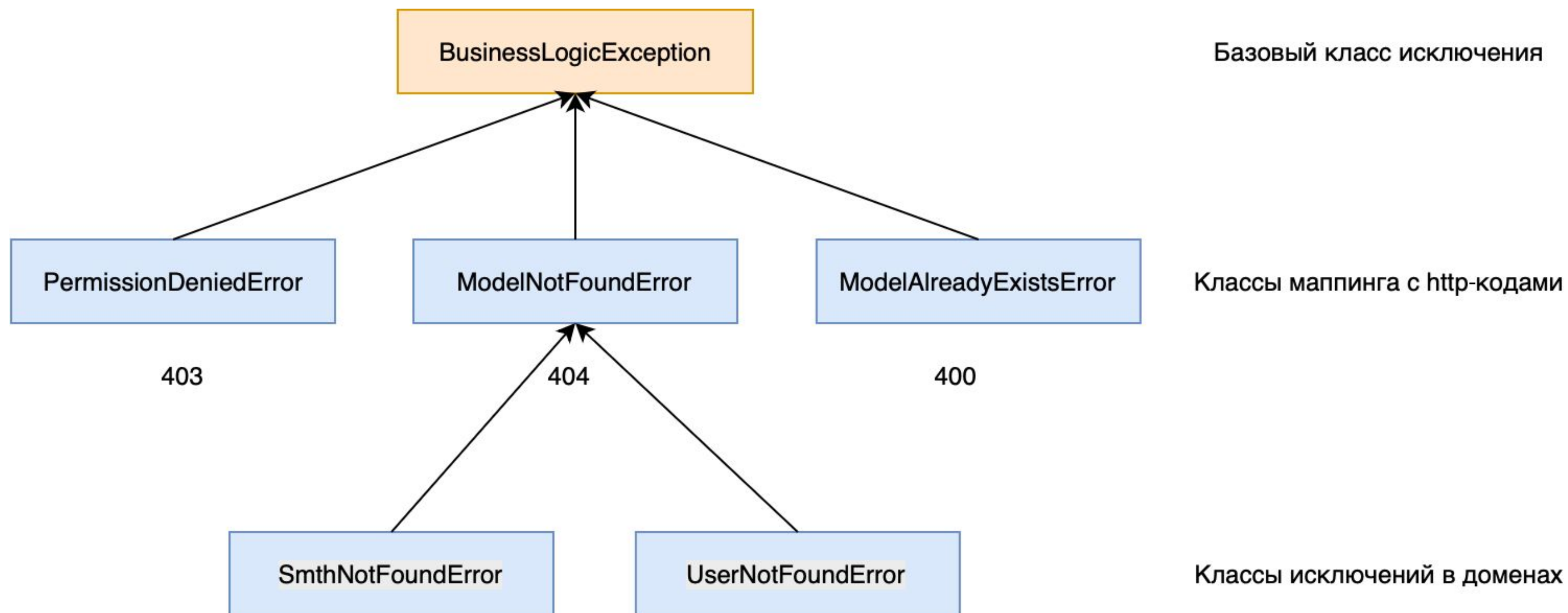
Регистрируем в FastAPI exception handlers

точка

```
def use_exceptions_handlers(app: FastAPI, settings: CoreSettingsSchema) -> None:
    """Регистрируем глобальные обработчики исключений."""
    app.exception_handler(BusinessLogicException)(partial(business_logic_exception_handler, settings))
    app.exception_handler(PermissionDeniedError)(partial(permission_denied_error_handler, settings))
    app.exception_handler(ModelNotFoundError)(partial(model_not_found_error_handler, settings))
    app.exception_handler(ModelAlreadyExistsError)(partial(model_already_exists_error_handler, settings))
```

Иерархия исключений

точка

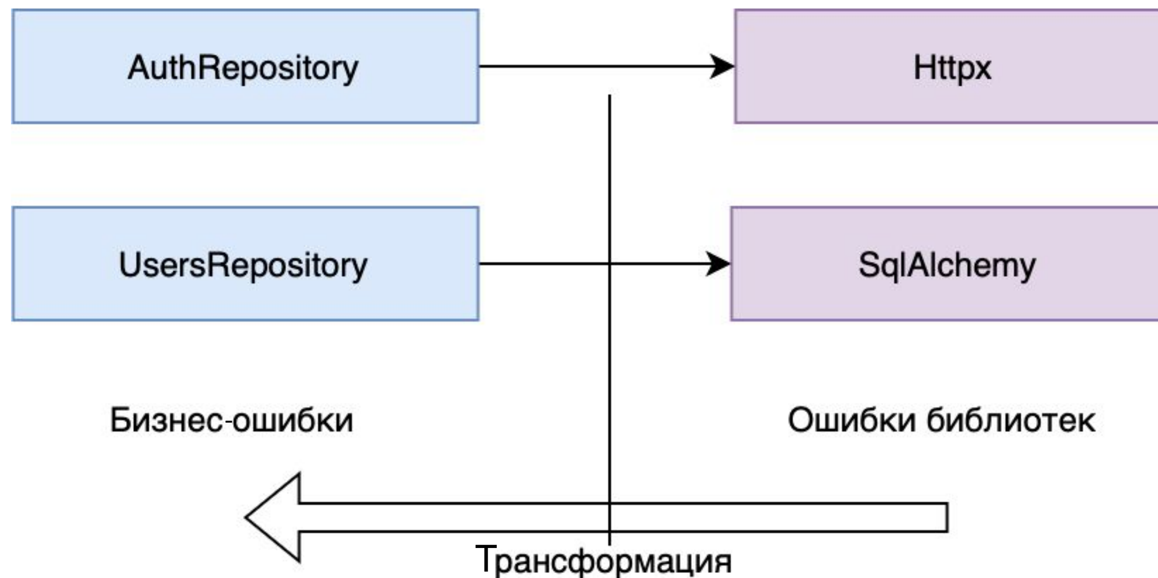


С бизнес-ошибками понятно, что делать с ошибками в репозиториях?

Трансформация исключений

точка

Ловим ошибки внешних библиотек
и трансформируем в свои



Ловим исключение и трансформируем в своё

точка

```python
class DbCrudRepositoryBase(
    Generic[ModelBaseType, ReadSchemaBaseType, CreateSchemaBaseType, UpdateSchemaBaseType, IdType]
):
    """Базовый репозиторий для выполнения CRUD операций над моделями в базе данных."""

    async def create(self: Self, create_object: CreateSchemaBaseType) -> ReadSchemaBaseType:
        """
        Создаем модель.
        """
        async with self.session_manager.get_session() as s:
            try:
                model_type = self.create_models_mapping[type(create_object)]
                create_dict = self.dump_create_object(create_object)
                return (await self.bulk_create_with_model_type(model_type, [create_dict], s))[0]
            except IntegrityError as integrity_error:
                raise ModelIntegrityError(self.model_type, ModelActionEnum.INSERT) from integrity_error
```

Ограничения чистой архитектуры

- Высокий порог входа — долго объяснять, но потом «не оторвать»
- Избыточно для простых проектов — не используем там, где не нужно
- Частые изменения? Риск накладных расходов — обёртки и маппинги усложняют поддержку
- Сложная интеграция сторонних сервисов: например, APScheduler

«Если вам кажется,
что хорошая архитектура
стоит дорого, попробуйте
плохую».

© Мартин Фаулер

точка для предпринимателей и предприятий

Спасибо!

Телеграм: @luferovvs

Телеграм-блог: @vityawritecode

YouTube: @luferov