



Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Информационная безопасность»

ОТЧЕТ

по лабораторной работе № 7

по курсу «Искусственный интеллект»

на тему: «FastAPI, разработка API приложения»

Вариант № 6

Студент ИУ8-13М
(Группа)

(Подпись, дата)

Савватеев А. Э.
(И. О. Фамилия)

Преподаватель

(Подпись, дата)

Зотов М. В.
(И. О. Фамилия)

2025 г.

СОДЕРЖАНИЕ

1	ЦЕЛЬ РАБОТЫ	3
2	ТРЕБОВАНИЯ ЛАБОРАТОРНОЙ РАБОТЫ	4
3	ПОСТАНОВКА ЗАДАЧИ	5
3.1	Вариант 6	5
4	ХОД РАБОТЫ	6
4.1	Архитектура приложения	6
4.2	Инфраструктура	6
4.3	Модели данных	6
4.3.1	SQLAlchemy ORM модели	6
4.3.2	Pydantic модели	7
4.4	Бизнес-логика	7
4.5	API эндпоинты	7
4.6	Главное приложение	8
5	РЕЗУЛЬТАТЫ РАБОТЫ	9
5.1	Запуск приложения	9
5.2	Примеры работы	9
5.3	Валидация данных	9
6	ЗАКЛЮЧЕНИЕ	11
	ПРИЛОЖЕНИЕ А Исходный код	12
A.1	Основные файлы	12
A.2	Модели	19
A.3	Операции с базой данных	23

1 ЦЕЛЬ РАБОТЫ

Реализовать REST API для функционала приложения по варианту с использованием современного стека технологий Python: FastAPI, SQLAlchemy 2.0, Pydantic v2, asyncpg.

2 ТРЕБОВАНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

- API использует асинхронное подключение к PostgreSQL через asyncpg
- API строит ORM-модели на SQLAlchemy 2.0
- API применяет валидацию и сериализацию через Pydantic (v2)
- REST API доступен через FastAPI
- API запускается через uvicorn

3 ПОСТАНОВКА ЗАДАЧИ

3.1 Вариант 6

Описание: API для ведения личной библиотеки — добавление книг, отслеживание прогресса чтения.

Основные сущности:

- `book` (`id`, `title`, `author`, `isbn`, `total_pages`)
- `readingEntry` (`id`, `book_id`, `user_id`, `current_page`, `updated_at`)
- `user` (`id`, `name`)

Эндпоинты:

- добавление книги
- обновление текущей страницы
- получение списка непрочитанных/прочитанных книг

4 ХОД РАБОТЫ

4.1 Архитектура приложения

Разработанное приложение следует многоуровневой архитектуре с четким разделением ответственности:

- **main.py** — точка входа, инициализация FastAPI приложения
- **router_library.py** — определение REST API эндпоинтов
- **models/store_models.py** — Pydantic модели для валидации данных
- **models/database_models.py** — SQLAlchemy ORM модели
- **operations/database_operations.py** — CRUD операции
- **operations/database_migrations.py** — миграции БД

4.2 Инфраструктура

PostgreSQL запущен в Docker контейнере с оптимизированными настройками: 1000 одновременных подключений, 256MB shared buffers, 768MB effective cache size. Healthcheck проверяет готовность БД каждые 30 секунд.

4.3 Модели данных

4.3.1 SQLAlchemy ORM модели

Разработаны три ORM-модели с использованием декларативного стиля SQLAlchemy 2.0:

- **UserOrm** — пользователи (id, name)
- **BookOrm** — книги (id, title, author, isbn, total_pages)
- **ReadingEntryOrm** — записи чтения (id, book_id, user_id, current_page, updated_at)

Настроены каскадные удаления и Foreign Key для ссылочной целостности. Временные метки хранятся с timezone.

4.3.2 Pydantic модели

Для валидации данных созданы Pydantic v2 модели:

- **UserAdd, UserGet** — для создания и получения пользователей
- **BookAdd, BookGet** — для работы с книгами
- **ReadingEntryAdd, ReadingEntryUpdate, ReadingEntryWithBookInfo** — для записей чтения

Применены валидаторы: `total_pages > 0`, `current_page >= 0`, проверка timezone в datetime.

4.4 Бизнес-логика

Реализованы три асинхронных класса-workflow:

- **UserWorkflow** — управление пользователями (создание, получение по ID, удаление)
- **BookWorkflow** — управление книгами (CRUD операции, поиск по автору через `ilike`)
- **ReadingEntryWorkflow** — управление прогрессом чтения:
 - добавление записи с валидацией существования книги и корректности номера страницы
 - обновление прогресса с проверкой ограничений
 - получение записей с `eager loading (selectinload)` для избежания N+1 проблемы
 - фильтрация прочитанных/непрочитанных книг

4.5 API эндпоинты

Определены три роутера с REST API эндпоинтами:

Books Router (/books):

- **POST /** — добавление книги

- GET /{book_id} — получение книги по ID
- GET / — список всех книг
- GET /search/author/{author_name} — поиск по автору

Users Router (/users):

- POST / — создание пользователя
- GET /{user_id} — получение пользователя

Reading Router (/reading):

- POST / — создание записи чтения
- PUT /{user_id}/books/{book_id} — обновление прогресса
- GET /user/{user_id}/all — все записи пользователя
- GET /user/{user_id}/books/unfinished — неп прочитанные книги
- GET /user/{user_id}/books/finished — прочитанные книги
- DELETE /{user_id}/books/{book_id} — удаление записи

Реализована обработка ошибок: ValueError → 400, отсутствие ресурса → 404, системные ошибки → 500.

4.6 Главное приложение

FastAPI приложение инициализируется с автоматической документацией (Swagger UI: /docs, ReDoc: /redoc). Использован lifespan event handler для автоматического создания таблиц при запуске.

Базовые эндпоинты:

- GET / — информация об API
- GET /health — проверка работоспособности

Запуск через uvicorn с автоматической перезагрузкой: `uvicorn main:app -reload`

5 РЕЗУЛЬТАТЫ РАБОТЫ

5.1 Запуск приложения

1. Запуск PostgreSQL: `docker-compose up -d`
2. Установка зависимостей: `pip install -r requirements.txt`
3. Запуск API: `uvicorn main:app -reload`

Приложение доступно на `http://localhost:8000`, документация на `/docs` и `/redoc`.

5.2 Примеры работы

Реализованный API позволяет:

- Создавать пользователей (POST `/users/`)
- Добавлять книги с указанием автора, названия, ISBN и количества страниц (POST `/books/`)
- Начинать чтение книги (POST `/reading/`)
- Обновлять прогресс чтения с валидацией (PUT `/reading/{user_id}/books/{book_id}`)
- Получать списки прочитанных и непрочитанных книг (GET `/reading/user/{user_id}/books/finished|unfinished`)
- Искать книги по автору (GET `/books/search/author/{author_name}`)

5.3 Валидация данных

Rydantic автоматически проверяет корректность входных данных:

- Отклоняет некорректные значения (отрицательное количество страниц)
- Бизнес-логика проверяет, что текущая страница не превышает общее количество

- Валидация существования связанных сущностей (книга должна существовать)
- Возвращаются детальные сообщения об ошибках с правильными HTTP кодами

6 ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы успешно реализовано REST API для управления личной библиотекой.

Достигнутые результаты:

- Разработана многоуровневая архитектура с разделением ответственности
- Реализованы все требуемые эндпоинты для управления книгами, пользователями и прогрессом чтения
- Обеспечена валидация данных на уровне Pydantic и бизнес-логики
- Применены асинхронные операции с БД через SQLAlchemy 2.0 + asyncpg
- Настроена автоматическая документация API

Соответствие требованиям:

- Асинхронное подключение к PostgreSQL через asyncpg
- ORM-модели на SQLAlchemy 2.0
- Валидация через Pydantic v2
- REST API на FastAPI
- Запуск через uvicorn
- PostgreSQL в Docker

ПРИЛОЖЕНИЕ А

Исходный код

А.1 Основные файлы

Листинг А.1 – Главный файл приложения (main.py)

```
1 from contextlib import asynccontextmanager
2
3 from fastapi import FastAPI
4 from fastapi.responses import JSONResponse
5
6 from .operations.database_migrations import create_tables
7 from router_library import book_router, user_router,
   reading_router
8
9
10 @asynccontextmanager
11 async def lifespan(app: FastAPI):
12     print(" Приложение запускается...")
13
14     try:
15         await create_tables()
16         print(" Таблицы в БД готовы")
17     except Exception as e:
18         print(f" Ошибка при создании таблиц: {e}")
19         raise
20
21     yield
22
23     print(" Приложение останавливается...")
24
25
26 app = FastAPI(
27     title="Personal Library API",
28     description="API для ведения личной библиотеки и
   отслеживания прогресса чтения",
29     version="1.0.0",
30     docs_url="/docs",
31     redoc_url="/redoc",
32     lifespan=lifespan,
33 )
```

```

34
35
36 @app.get("/", tags=["Root"])
37 async def root():
38     return {
39         "message": "Welcome to Personal Library API",
40         "docs": "/docs",
41         "redoc": "/redoc",
42         "version": "1.0.0"
43     }
44
45
46 @app.get("/health", tags=["Health"])
47 async def health_check():
48     return {
49         "status": "healthy",
50         "api": "running"
51     }
52
53
54 app.include_router(book_router)
55 app.include_router(user_router)
56 app.include_router(reading_router)
57
58
59 @app.exception_handler(HTTPException)
60 async def http_exception_handler(request, exc):
61     return JSONResponse(
62         status_code=exc.status_code,
63         content={
64             "error": True,
65             "status_code": exc.status_code,
66             "detail": exc.detail
67         }
68     )
69
70
71 if __name__ == "__main__":
72     import uvicorn
73
74     uvicorn.run(

```

```

75         "main:app",
76         host="0.0.0.0",
77         port=8000,
78         reload=True,
79         log_level="info"
80     )

```

Листинг А.2 – Определение API эндпоинтов (router_library.py)

```

1  from fastapi import APIRouter, Depends, HTTPException, status
2  from typing import List, Optional
3
4  from operations.database_operations import (
5      BookWorkflow, UserWorkflow, ReadingEntryWorkflow
6  )
7  from models.store_models import (
8      BookAdd, BookGet,
9      UserAdd, UserGet,
10     ReadingEntryAdd, ReadingEntryUpdate, ReadingEntryWithBookInfo
11 )
12
13
14 book_router = APIRouter(
15     prefix="/books",
16     tags=["Книги"],
17 )
18
19
20 @book_router.post("/", response_model=BookGet,
21     status_code=status.HTTP_201_CREATED)
22 async def add_book(book: BookAdd = Depends()) -> BookGet:
23     try:
24         book_id = await BookWorkflow.add_book(book)
25         created_book = await BookWorkflow.get_book_by_id(book_id)
26         return created_book
27     except Exception as e:
28         raise HTTPException(
29             status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
30             detail=f"Failed to add book: {str(e)}"
31         )
32
33 @book_router.get("/{book_id}", response_model=BookGet)

```

```

34 async def get_book(book_id: int) -> BookGet:
35     book = await BookWorkflow.get_book_by_id(book_id)
36     if not book:
37         raise HTTPException(
38             status_code=status.HTTP_404_NOT_FOUND,
39             detail=f"Book with ID {book_id} not found"
40         )
41     return book
42
43
44 @book_router.get("/", response_model=List[BookGet])
45 async def get_all_books() -> List[BookGet]:
46     return await BookWorkflow.get_all_books()
47
48
49 @book_router.get("/search/author/{author_name}",
50                 response_model=List[BookGet])
51 async def get_books_by_author(author_name: str) -> List[BookGet]:
52     return await BookWorkflow.get_books_by_author(author_name)
53
54
55 user_router = APIRouter(
56     prefix="/users",
57     tags=["Пользователи"],
58 )
59
60 @user_router.post("/", response_model=UserGet,
61                  status_code=status.HTTP_201_CREATED)
62 async def add_user(user: UserAdd = Depends()) -> UserGet:
63     try:
64         user_id = await UserWorkflow.add_user(user)
65         created_user = await UserWorkflow.get_user_by_id(user_id)
66         return created_user
67     except Exception as e:
68         raise HTTPException(
69             status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
70             detail=f"Failed to add user: {str(e)}"
71         )
72

```

```

73 @user_router.get("/{user_id}", response_model=UserGet)
74 async def get_user(user_id: int) -> UserGet:
75     user = await UserWorkflow.get_user_by_id(user_id)
76     if not user:
77         raise HTTPException(
78             status_code=status.HTTP_404_NOT_FOUND,
79             detail=f"User with ID {user_id} not found"
80         )
81     return user
82
83
84 reading_router = APIRouter(
85     prefix="/reading",
86     tags=["Записи чтения"],
87 )
88
89
90 @reading_router.post("/",
91     response_model=ReadingEntryWithBookInfo,
92     status_code=status.HTTP_201_CREATED)
91 async def add_reading_entry(entry: ReadingEntryAdd = Depends())
92     -> ReadingEntryWithBookInfo:
93     try:
94         entry_id = await
95             ReadingEntryWorkflow.add_reading_entry(entry)
96
97         entries = await
98             ReadingEntryWorkflow.get_reading_entries_by_user(entry.user_id)
99         created_entry = next((e for e in entries if e.id ==
100             entry_id), None)
101
102         if not created_entry:
103             raise HTTPException(
104                 status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
105                 detail="Failed to retrieve created reading entry"
106             )
107
108         return created_entry
109     except ValueError as ve:
110         raise HTTPException(
111             status_code=status.HTTP_400_BAD_REQUEST,

```

```

108         detail=str(ve)
109     )
110     except Exception as e:
111         raise HTTPException(
112             status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
113             detail=f"Failed to add reading entry: {str(e)}"
114         )
115
116
117 @reading_router.put("/{user_id}/books/{book_id}",
118                     response_model=ReadingEntryWithBookInfo)
119 async def update_reading_progress(
120     user_id: int,
121     book_id: int,
122     update_data: ReadingEntryUpdate = Depends()
123 ) -> ReadingEntryWithBookInfo:
124     try:
125         updated_entry = await
126             ReadingEntryWorkflow.update_reading_progress(
127                 user_id=user_id,
128                 book_id=book_id,
129                 update_data=update_data
130             )
131
132         if not updated_entry:
133             raise HTTPException(
134                 status_code=status.HTTP_404_NOT_FOUND,
135                 detail=f"Reading entry for user {user_id} and
136                     book {book_id} not found"
137             )
138
139         entries = await
140             ReadingEntryWorkflow.get_reading_entries_by_user(user_id)
141         full_entry = next((e for e in entries if e.id ==
142             updated_entry.id), None)
143
144         return full_entry
145     except ValueError as ve:
146         raise HTTPException(
147             status_code=status.HTTP_400_BAD_REQUEST,
148             detail=str(ve)

```

```

144         )
145
146
147 @reading_router.get("/user/{user_id}/all",
148     response_model=List[ReadingEntryWithBookInfo])
149 async def get_user_reading_entries(user_id: int) ->
150     List[ReadingEntryWithBookInfo]:
151     entries = await
152         ReadingEntryWorkflow.get_reading_entries_by_user(user_id)
153     if not entries:
154         return []
155     return entries
156
157
158 @reading_router.get("/user/{user_id}/books/unfinished",
159     response_model=List[ReadingEntryWithBookInfo])
160 async def get_unfinished_books(user_id: int) ->
161     List[ReadingEntryWithBookInfo]:
162     return await
163         ReadingEntryWorkflow.get_unfinished_books(user_id)
164
165
166 @reading_router.get("/user/{user_id}/books/finished",
167     response_model=List[ReadingEntryWithBookInfo])
168 async def get_finished_books(user_id: int) ->
169     List[ReadingEntryWithBookInfo]:
170     return await ReadingEntryWorkflow.get_finished_books(user_id)
171
172
173 @reading_router.delete("/{user_id}/books/{book_id}",
174     status_code=status.HTTP_204_NO_CONTENT)
175 async def delete_reading_entry(user_id: int, book_id: int):
176     deleted = await
177         ReadingEntryWorkflow.delete_reading_entry(user_id,
178             book_id)
179     if not deleted:
180         raise HTTPException(
181             status_code=status.HTTP_404_NOT_FOUND,
182             detail=f"Reading entry for user {user_id} and book
183                 {book_id} not found"
184         )
185

```

A.2 Модели

Листинг A.3 – SQLAlchemy ORM модели (database_models.py)

```
1 from sqlalchemy.orm import DeclarativeBase, Mapped,  
    mapped_column, relationship  
2 from sqlalchemy import DateTime, BIGINT, String, ForeignKey,  
    Integer  
3 from datetime import datetime  
4 from zoneinfo import ZoneInfo  
5  
6  
7 def utc_now() -> datetime:  
8     return datetime.now(ZoneInfo("UTC"))  
9  
10  
11 class TableModel(DeclarativeBase):  
12     pass  
13  
14  
15 class UserOrm(TableModel):  
16     __tablename__ = "user"  
17     __table_args__ = {"schema": "public"}  
18  
19     id: Mapped[int] = mapped_column(BIGINT, primary_key=True,  
        autoincrement=True)  
20     name: Mapped[str] = mapped_column(String(255),  
        nullable=False)  
21  
22     reading_entries: Mapped[list["ReadingEntryOrm"]] =  
        relationship(  
23         "ReadingEntryOrm",  
24         back_populates="user",  
25         cascade="all, delete-orphan"  
26     )  
27  
28  
29 class BookOrm(TableModel):  
30     __tablename__ = "book"  
31     __table_args__ = {"schema": "public"}  
32  
33     id: Mapped[int] = mapped_column(BIGINT, primary_key=True,
```

```

    autoincrement=True)
34 title: Mapped[str] = mapped_column(String(500),
    nullable=False)
35 author: Mapped[str] = mapped_column(String(255),
    nullable=False)
36 isbn: Mapped[str | None] = mapped_column(String(20),
    nullable=True)
37 total_pages: Mapped[int] = mapped_column(Integer,
    nullable=False)
38
39 reading_entries: Mapped[list["ReadingEntryOrm"]] =
    relationship(
40     "ReadingEntryOrm",
41     back_populates="book",
42     cascade="all, delete-orphan"
43 )
44
45
46 class ReadingEntryOrm(TableModel):
47     __tablename__ = "reading_entry"
48     __table_args__ = {"schema": "public"}
49
50     id: Mapped[int] = mapped_column(BIGINT, primary_key=True,
        autoincrement=True)
51     book_id: Mapped[int] = mapped_column(BIGINT,
        ForeignKey("public.book.id"), nullable=False)
52     user_id: Mapped[int] = mapped_column(BIGINT,
        ForeignKey("public.user.id"), nullable=False)
53     current_page: Mapped[int] = mapped_column(Integer,
        nullable=False, default=0)
54     updated_at: Mapped[datetime] = mapped_column(
55         DateTime(timezone=True),
56         default=utc_now,
57         nullable=False)
58 )
59
60     book: Mapped["BookOrm"] = relationship("BookOrm",
        back_populates="reading_entries")
61     user: Mapped["UserOrm"] = relationship("UserOrm",
        back_populates="reading_entries")

```

Листинг А.4 – Pydantic модели для валидации (store_models.py)

```

1 from pydantic import BaseModel, AfterValidator, Field,
   ConfigDict, RootModel
2 from typing import Annotated, Optional, List
3 from datetime import datetime
4 from zoneinfo import ZoneInfo
5
6
7 def validate_aware(dt: datetime) -> datetime:
8     if dt.tzinfo is None:
9         raise ValueError("datetime must include timezone
10                             information")
11     return dt
12
13 AwareDatetime = Annotated[datetime,
14                             AfterValidator(validate_aware)]
15
16 def default_utc_time() -> datetime:
17     return datetime.now(ZoneInfo("UTC"))
18
19
20 class UserBase(BaseModel):
21     name: str = Field(description="Имя пользователя")
22
23
24 class UserAdd(UserBase):
25     pass
26
27
28 class UserGet(UserBase):
29     id: int
30
31     model_config =
32         ConfigDict(from_attributes=True, extra='forbid')
33
34 class BookBase(BaseModel):
35     title: str = Field(description="Название книги")
36     author: str = Field(description="Автор книги")
37     isbn: Optional[str] = Field(None, description="ISBN код

```

```

        книги")
38     total_pages: int = Field(gt=0, description="Общее количество
        страниц в книге")
39
40
41 class BookAdd(BookBase):
42     pass
43
44
45 class BookGet(BookBase):
46     id: int
47
48     model_config =
49         ConfigDict(from_attributes=True, extra='forbid')
50
51 class ReadingEntryBase(BaseModel):
52     book_id: int = Field(description="ID книги")
53     user_id: int = Field(description="ID пользователя")
54     current_page: int = Field(ge=0, description="Текущая
        страница чтения")
55     updated_at: Optional[AwareDatetime] = Field(
56         default_factory=default_utc_time,
57         description="Дата последнего обновления прогресса чтения"
58     )
59
60
61 class ReadingEntryAdd(ReadingEntryBase):
62     pass
63
64
65 class ReadingEntryUpdate(BaseModel):
66     current_page: int = Field(ge=0, description="Текущая
        страница чтения")
67     updated_at: Optional[AwareDatetime] = Field(
68         default_factory=default_utc_time,
69         description="Дата обновления прогресса чтения"
70     )
71
72
73 class ReadingEntryGet(ReadingEntryBase):

```

```

74     id: int
75
76     model_config =
77         ConfigDict(from_attributes=True, extra='forbid')
78
79 class ReadingEntryWithBookInfo(ReadingEntryGet):
80     book: BookGet
81     is_finished: bool = Field(description="Прочитана ли книга
82         полностью")
83
84 class BookListGet(RootModel[List[BookGet]]):
85     model_config =
86         ConfigDict(from_attributes=True, extra='forbid')
87
88 class
89     ReadingEntryListGet(RootModel[List[ReadingEntryWithBookInfo]]):
90     model_config =
91         ConfigDict(from_attributes=True, extra='forbid')
92
93 class UserListGet(RootModel[List[UserGet]]):
94     model_config =
95         ConfigDict(from_attributes=True, extra='forbid')

```

A.3 Операции с базой данных

Листинг A.5 – CRUD операции и бизнес-логика (database_operations.py)

```

1 from sqlalchemy.ext.asyncio import async_sessionmaker,
    create_async_engine
2 from sqlalchemy import select, delete, update, and_
3 from sqlalchemy.orm import selectinload
4 from typing import List, Optional
5
6 from ..models.database_models import UserOrm, BookOrm,
    ReadingEntryOrm
7 from ..models.store_models import (
8     UserAdd, UserGet,
9     BookAdd, BookGet,
10    ReadingEntryAdd, ReadingEntryUpdate, ReadingEntryGet,

```

```

        ReadingEntryWithBookInfo
11 )
12
13
14 engine = create_async_engine(
15     "postgresql+asyncpg://postgres:postgres@localhost:5432/postgres",
16     echo=True
17 )
18 new_session = async_sessionmaker(engine, expire_on_commit=False)
19
20
21 class UserWorkflow:
22
23     @classmethod
24     async def add_user(cls, user: UserAdd) -> int:
25         async with new_session() as session:
26             new_user = UserOrm(name=user.name)
27             session.add(new_user)
28             await session.flush()
29             await session.commit()
30             return new_user.id
31
32     @classmethod
33     async def get_user_by_id(cls, user_id: int) ->
34         Optional[UserGet]:
35         async with new_session() as session:
36             query = select(UserOrm).where(UserOrm.id == user_id)
37             result = await session.execute(query)
38             user = result.scalar_one_or_none()
39
40             if user:
41                 return UserGet.model_validate(user)
42             return None
43
44     @classmethod
45     async def get_all_users(cls) -> List[UserGet]:
46         async with new_session() as session:
47             query = select(UserOrm)
48             result = await session.execute(query)
49             users = result.scalars().all()

```

```

50         return [UserGet.model_validate(user) for user in
51                 users]
52
53     @classmethod
54     async def delete_user(cls, user_id: int) -> bool:
55         async with new_session() as session:
56             query = delete(UserOrm).where(UserOrm.id == user_id)
57             result = await session.execute(query)
58             await session.commit()
59             return result.rowcount > 0
60
61 class BookWorkflow:
62
63     @classmethod
64     async def add_book(cls, book: BookAdd) -> int:
65         async with new_session() as session:
66             new_book = BookOrm(
67                 title=book.title,
68                 author=book.author,
69                 isbn=book.isbn,
70                 total_pages=book.total_pages
71             )
72             session.add(new_book)
73             await session.flush()
74             await session.commit()
75             return new_book.id
76
77     @classmethod
78     async def get_book_by_id(cls, book_id: int) ->
79         Optional[BookGet]:
80         async with new_session() as session:
81             query = select(BookOrm).where(BookOrm.id == book_id)
82             result = await session.execute(query)
83             book = result.scalar_one_or_none()
84
85             if book:
86                 return BookGet.model_validate(book)
87             return None
88
89     @classmethod

```

```

89     async def get_all_books(cls) -> List[BookGet]:
90         async with new_session() as session:
91             query = select(BookOrm)
92             result = await session.execute(query)
93             books = result.scalars().all()
94
95             return [BookGet.model_validate(book) for book in
96                     books]
97
98     @classmethod
99     async def get_books_by_author(cls, author: str) ->
100     List[BookGet]:
101         async with new_session() as session:
102             query =
103                 select(BookOrm).where(BookOrm.author.ilike(f"%{author}%"))
104             result = await session.execute(query)
105             books = result.scalars().all()
106
107             return [BookGet.model_validate(book) for book in
108                     books]
109
110     @classmethod
111     async def delete_book(cls, book_id: int) -> bool:
112         async with new_session() as session:
113             query = delete(BookOrm).where(BookOrm.id == book_id)
114             result = await session.execute(query)
115             await session.commit()
116             return result.rowcount > 0
117
118 class ReadingEntryWorkflow:
119
120     @classmethod
121     async def add_reading_entry(cls, entry: ReadingEntryAdd) ->
122     int:
123         async with new_session() as session:
124             book_query = select(BookOrm).where(BookOrm.id ==
125                 entry.book_id)
126             book_result = await session.execute(book_query)
127             book = book_result.scalar_one_or_none()

```

```

124         if not book:
125             raise ValueError(f"Book with ID {entry.book_id}
126                               not found")
127
128         if entry.current_page > book.total_pages:
129             raise ValueError(
130                 f"Current page ({entry.current_page}) cannot
131                 exceed "
132                 f"total pages ({book.total_pages})"
133             )
134
135         new_entry = ReadingEntryOrm(
136             book_id=entry.book_id,
137             user_id=entry.user_id,
138             current_page=entry.current_page
139         )
140         session.add(new_entry)
141         await session.flush()
142         await session.commit()
143         return new_entry.id
144
145     @classmethod
146     async def update_reading_progress(
147         cls,
148         user_id: int,
149         book_id: int,
150         update_data: ReadingEntryUpdate
151     ) -> Optional[ReadingEntryGet]:
152         async with new_session() as session:
153             book_query = select(BookOrm).where(BookOrm.id ==
154                                                 book_id)
155             book_result = await session.execute(book_query)
156             book = book_result.scalar_one_or_none()
157
158             if not book:
159                 raise ValueError(f"Book with ID {book_id} not
160                                   found")
161
162             if update_data.current_page > book.total_pages:
163                 raise ValueError(
164                     f"Current page ({update_data.current_page})

```

```

161         cannot exceed "
162         f"total pages ({book.total_pages})"
163     )
164     query = (
165         update(ReadingEntryOrm)
166         .where(
167             and_(
168                 ReadingEntryOrm.user_id == user_id,
169                 ReadingEntryOrm.book_id == book_id
170             )
171         )
172         .values(
173             current_page=update_data.current_page,
174             updated_at=update_data.updated_at
175         )
176         .returning(ReadingEntryOrm)
177     )
178
179     result = await session.execute(query)
180     await session.commit()
181
182     updated_entry = result.scalar_one_or_none()
183     if updated_entry:
184         return
185         ReadingEntryGet.model_validate(updated_entry)
186     return None
187
188 @classmethod
189 async def get_reading_entries_by_user(cls, user_id: int) ->
190     List[ReadingEntryWithBookInfo]:
191     async with new_session() as session:
192         query = (
193             select(ReadingEntryOrm)
194             .where(ReadingEntryOrm.user_id == user_id)
195             .options(selectinload(ReadingEntryOrm.book))
196         )
197         result = await session.execute(query)
198         entries = result.scalars().all()
199
200     response = []

```

```

199         for entry in entries:
200             entry_with_book = ReadingEntryWithBookInfo(
201                 id=entry.id,
202                 book_id=entry.book_id,
203                 user_id=entry.user_id,
204                 current_page=entry.current_page,
205                 updated_at=entry.updated_at,
206                 book=BookGet.model_validate(entry.book),
207                 is_finished=entry.current_page >=
                    entry.book.total_pages
208             )
209             response.append(entry_with_book)
210
211         return response
212
213     @classmethod
214     async def get_unfinished_books(cls, user_id: int) ->
        List[ReadingEntryWithBookInfo]:
215         async with new_session() as session:
216             query = (
217                 select(ReadingEntryOrm)
218                 .where(ReadingEntryOrm.user_id == user_id)
219                 .options(selectinload(ReadingEntryOrm.book))
220             )
221             result = await session.execute(query)
222             entries = result.scalars().all()
223
224             response = []
225             for entry in entries:
226                 if entry.current_page < entry.book.total_pages:
227                     entry_with_book = ReadingEntryWithBookInfo(
228                         id=entry.id,
229                         book_id=entry.book_id,
230                         user_id=entry.user_id,
231                         current_page=entry.current_page,
232                         updated_at=entry.updated_at,
233                         book=BookGet.model_validate(entry.book),
234                         is_finished=False
235                     )
236                     response.append(entry_with_book)
237

```

```

238         return response
239
240     @classmethod
241     async def get_finished_books(cls, user_id: int) ->
242         List[ReadingEntryWithBookInfo]:
243         async with new_session() as session:
244             query = (
245                 select(ReadingEntryOrm)
246                 .where(ReadingEntryOrm.user_id == user_id)
247                 .options(selectinload(ReadingEntryOrm.book))
248             )
249             result = await session.execute(query)
250             entries = result.scalars().all()
251
252             response = []
253             for entry in entries:
254                 if entry.current_page >= entry.book.total_pages:
255                     entry_with_book = ReadingEntryWithBookInfo(
256                         id=entry.id,
257                         book_id=entry.book_id,
258                         user_id=entry.user_id,
259                         current_page=entry.current_page,
260                         updated_at=entry.updated_at,
261                         book=BookGet.model_validate(entry.book),
262                         is_finished=True
263                     )
264                     response.append(entry_with_book)
265
266             return response
267
268     @classmethod
269     async def delete_reading_entry(cls, user_id: int, book_id:
270         int) -> bool:
271         async with new_session() as session:
272             query = delete(ReadingEntryOrm).where(
273                 and_(
274                     ReadingEntryOrm.user_id == user_id,
275                     ReadingEntryOrm.book_id == book_id
276                 )
277             )
278             result = await session.execute(query)

```

```
277         await session.commit()
278         return result.rowcount > 0
```

Листинг А.6 – Миграции базы данных (database_migrations.py)

```
1 from sqlalchemy.ext.asyncio import create_async_engine
2 from app.models.database_models import TableModel
3
4
5 engine =
6     create_async_engine("postgresql+asyncpg://postgres:postgres@localhost")
7
8 async def create_tables():
9     async with engine.begin() as conn:
10         await conn.run_sync(TableModel.metadata.create_all)
11         print("Таблицы успешно созданы в БД")
12
13
14 async def drop_tables():
15     async with engine.begin() as conn:
16         await conn.run_sync(TableModel.metadata.drop_all)
17         print("Таблицы успешно удалены из БД")
```