
Astronomy API на SQLModel

Содержание проекта

Полнофункциональное API для управления базой данных астрономических объектов, построенное на **FastAPI** и **SQLModel**.

Что такое SQLModel?

SQLModel - это библиотека от создателя FastAPI, которая объединяет:

- **SQLAlchemy** (ORM для баз данных)
- **Pydantic** (валидация данных)

В одном классе! Это упрощает код и уменьшает дублирование.

Функциональность

Содержание проекта

Основные возможности:

- **Управление небесными телами**: планеты, звезды, галактики, туманности и др.
- **Управление астрономами**: информация об ученых и их достижениях
- **Запись наблюдений**: связь между астрономами и небесными телами

-
- **Расширенный поиск**: фильтрация по множеству параметров
 - **Статистика**: аналитика по данным
 - **Аутентификация**: регистрация, вход, JWT токены
-

Технологии:

- **FastAPI** 0.109.0 - современный веб-фреймворк
 - **SQLModel** 0.0.14 - ORM + валидация в одном пакете
 - **Pydantic** 2.5.3 - валидация данных
 - **PostgreSQL / SQLite** - база данных
 - **JWT** - токены для аутентификации
-

📁 Структура проекта

```
AstroObservatory/
├── app/
│   ├── main.py # Точка входа
│   ├── database.py # Конфигурация БД
│   ├── models/ # Модели SQLModel
│   │   ├── base.py # Базовый миксин
│   │   ├── celestial_body.py
│   │   ├── astronomer.py
│   │   ├── observation.py
│   │   └── user.py
│   ├── routers/ # Маршруты
│   │   ├── celestial_bodies.py
│   │   ├── astronomers.py
│   │   ├── observations.py
│   │   └── auth.py
│   └── services/ # Сервисы
│       └── search.py
└── .env # Переменные окружения
└── requirements.txt # Зависимости
└── README.md # Документация
```

Команды

Содержание проекта

Запуск приложения

```
uvicorn app.main:app --reload
```

API документация

После запуска приложения доступна автоматическая документация:

- **Swagger UI:** <http://localhost:8000/docs>
- **ReDoc:** <http://localhost:8000/redoc>
- **OpenAPI JSON:** <http://localhost:8000/openapi.json>

🔑 Примеры использования

1. Регистрация пользователя:

```
curl -X POST "http://localhost:8000/auth/register" \
-H "Content-Type: application/json" \
-d '{
    "username": "astronomer1",
    "email": "astronomer@example.com",
    "full_name": "Иван Иванов",
    "password": "SecurePass123"
}'
```

Получение токена

```
curl -X POST "http://localhost:8000/auth/token" \
-H "Content-Type: application/x-www-form-urlencoded" \
-d "username=astronomer1&password=SecurePass123"
```

Создание небесного тела:

```
curl -X POST "http://localhost:8000/celestial-bodies/" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer YOUR_TOKEN" \
-d '{
    "name": "Солнце",
    "type": "star",
    "description": "Звезда главной последовательности",
    "mass": 1.0,
    "radius": 1.0,
    "temperature": 5778,
    "distance_from_earth": 0.0000158,
    "spectral_class": "G",
    "absolute_magnitude": 4.83,
    "apparent_magnitude": -26.74
}'
```

Создание астронома

```
curl -X POST "http://localhost:8000/astronomers/" \
-H "Content-Type: application/json" \
-d '{
    "first_name": "Николай",
    "last_name": "Коперник",
    "birth_date": "1473-02-19",
    "death_date": "1543-05-24",
    "nationality": "Польский",
    "institution": "Фромборкский собор",
    "notable_discoveries": "Гелиоцентрическая система мира"
}'
```

Создание наблюдения

```
curl -X POST "http://localhost:8000/observations/" \
-H "Content-Type: application/json" \
-d '{
    "astronomer_id": 1,
    "celestial_body_id": 1,
    "observation_date": "2024-01-15T20:00:00",
    "location": "Обсерватория Паломар",
    "equipment": "Телескоп Хейла",
    "duration_hours": 3.5,
    "notes": "Отличные условия наблюдения"
}'
```

Статистика

API предоставляет эндпоинты для получения статистики:

- </celestial-bodies/statistics> - статистика по небесным телам
- </astronomers/statistics> - статистика по астрономам
- </observations/statistics> - статистика по наблюдениям

Гарантии

Этот проект:

- Полностью рабочий (без ошибок)
- Использует современный синтаксис SQLModel
- Содержит подробные комментарии
- Готов к запуску и изучению
- Использует асинхронное программирование

Изучение

Каждый файл содержит подробные комментарии на русском языке, объясняющие:

- Что делает код
- Как это работает
- Зачем это нужно

Идеально подходит для изучения:

- **FastAPI** - маршруты, зависимости, обработка ошибок
- **SQLModel** - модели, отношения, запросы
- **Pydantic** - валидация, схемы
- **Асинхронное программирование** - `async/await`
- **Базы данных** - SQLite/PostgreSQL
- **JWT** - аутентификация

Модули проекта

Содержание проекта

app/database.py	<p>Конфигурация базы данных и сессии.</p> <p>Использует SQLModel (SQLAlchemy 2.0 + Pydantic в одном пакете).</p>
app/models/__init__.py	<p>Модули моделей базы данных.</p> <p>SQLModel объединяет SQLAlchemy модели и Pydantic схемы в одном классе.</p>
app/models/base.py	<p>Базовый миксин для всех моделей.</p> <p>Добавляет временные метки к моделям</p>
app/models/celestial_body.py	<p>Модель небесных тел (планеты, звезды, галактики)</p>
app/models/astronomer.py	<p>Модель астрономов.</p> <p>Содержит информацию об ученых и их достижениях.</p>
app/models/observation.py	<p>Модель наблюдений астрономов за небесными телами.</p> <p>Промежуточная таблица для связи многие-ко-многим.</p>
app/models/user.py	<p>Модель пользователей для аутентификации.</p>
app/routers/init.py	<p>Маршруты (routers) приложения.</p> <p>Содержат все эндпоинты API.</p>
app/routers/celestial_bodies.py	<p>Маршрут для работы с небесными телами.</p>

<code>app/database.py</code>	Конфигурация базы данных и сессии. Использует SQLModel (SQLAlchemy 2.0 + Pydantic в одном пакете).
	Содержит все CRUD операции и дополнительные методы.
<code>app/routers/astronomers.py</code>	Маршрут для работы с астрономами
<code>app/routers/observations.py</code>	Маршрут для работы с наблюдениями.
<code>app/routers/auth.py</code>	Маршрут для аутентификации. Включает регистрацию, вход и работу с токенами JWT.
<code>app/services/init.py</code>	Сервисы приложения. Содержат бизнес-логику и вспомогательные функции.
<code>app/services/search.py</code>	Сервис для расширенного поиска и фильтрации.
<code>app/main.py</code>	Главный файл приложения FastAPI. Содержит конфигурацию и подключение маршрутов.
Команды	Рабочие команды проекта

app/database.py

Содержание

database.py

```
"""

```

Конфигурация базы данных и сессии.

Использует SQLModel (SQLAlchemy 2.0 + Pydantic в одном пакете).

```
"""

```

```
from sqlmodel import SQLModel
from sqlmodel.ext.asyncio.session import AsyncSession
from sqlalchemy.ext.asyncio import create_async_engine,
async_sessionmaker
from sqlalchemy.pool import NullPool
from typing import AsyncGenerator
import os
from dotenv import load_dotenv
```

```
# Загрузка переменных окружения из .env файла
load_dotenv()

# URL подключения к базе данных
DATABASE_URL = os.getenv("DATABASE_URL")

if not DATABASE_URL:
    raise RuntimeError(
        "❌ DATABASE_URL не задан!
        Создайте .env файл или установите переменную
окружения."
    )

# Создание асинхронного движка (engine)
engine = create_async_engine(
    DATABASE_URL,
    echo=True,                      # Показывать SQL запросы в консоли
    future=True,                     # Использовать новый стиль SQLAlchemy
    2.0,
    poolclass=NullPool      # Отключает пул соединений (для тестов)
)

# Создание фабрики сессий
AsyncSessionLocal = async_sessionmaker(
    engine,
    class_=AsyncSession,
    expire_on_commit=False,
    autoflush=False,
    autocommit=False
)

# Dependency для получения сессии базы данных
async def get_db() → AsyncGenerator[AsyncSession, None]:
    """
    Dependency для получения асинхронной сессии базы данных.
    """

    Использует контекстный менеджер для гарантии закрытия
    сессии.
```

Даже если произойдет ошибка, сессия будет закрыта.

Пример использования в маршруте:

```
@app.get("/items")
async def get_items(db: AsyncSession = Depends(get_db)):
    ...
"""
async with AsyncSessionLocal() as session:
    try:
        yield session
        await session.commit()
    except Exception:
        await session.rollback()
        raise
    finally:
        await session.close()
```

Функция для инициализации базы данных (создание таблиц)

```
async def init_db():
    """

```

Инициализация базы данных: создание всех таблиц.

ВАЖНО: В продакшене используйте Alembic для миграций!

```
"""
from app.models import (
    celestial_body,
    astronomer,
    observation,
    user
)

async with engine.begin() as conn:
    await conn.run_sync(SQLModel.metadata.create_all)
```

app/models/init.py

Содержание

`__init__.py`

```
"""
Модули моделей базы данных.

SQLModel объединяет SQLAlchemy модели и Pydantic схемы в одном
классе.

"""

from app.models.base import TimestampMixin
from app.models.celestial_body import CelestialBody, BodyType,
SpectralClass
from app.models.astronomer import Astronomer
from app.models.observation import Observation
from app.models.user import User

__all__ = [
    "TimestampMixin",
    "CelestialBody",
    "BodyType",
    "SpectralClass",
    "Astronomer",
    "Observation",
    "User"
]
```

app/models/base.py

Содержание

base.py

```
"""
Базовый миксин для всех моделей.

Добавляет временные метки к моделям.

"""

from sqlmodel import Field
from datetime import datetime, timezone
from typing import Optional
```

```
class TimestampMixin:  
    """  
    Миксин для автоматического добавления временных меток.  
  
    Добавляет поля created_at и updated_at в модели.  
    Эти поля автоматически заполняются при создании и обновлении  
    записей.  
  
    Пример использования:  
        class Article(SQLModel, TimestampMixin, table=True):  
            __tablename__ = "articles"  
            id: int = Field(default=None, primary_key=True)  
            """  
  
            # Время создания записи  
            created_at: datetime = Field(  
                default_factory=lambda: datetime.now(timezone.utc),  
                nullable=False,  
                sa_column_kwargs={"server_default": "CURRENT_TIMESTAMP"}  
            )  
  
            # Время последнего обновления записи  
            updated_at: datetime = Field(  
                default_factory=lambda: datetime.now(timezone.utc),  
                nullable=False,  
                sa_column_kwargs={"server_default": "CURRENT_TIMESTAMP",  
                "onupdate": "CURRENT_TIMESTAMP"}  
            )
```

app/models/celestial_body.py

Содержание

celestial_body.py

```
"""
```

Модель небесных тел (планеты, звезды, галактики).

Использует современный синтаксис SQLModel.

Содержит связи с другими моделями и вычисляемые свойства.

```
"""
```

```
from sqlmodel import SQLModel, Field, Relationship, Enum as
SQLEnum, Index
from typing import Optional, List, TYPE_CHECKING
from datetime import datetime
from enum import Enum as PyEnum

# Импорты для связей (избегаем циклических импортов)
if TYPE_CHECKING:
    from app.models.astronomer import Astronomer
    from app.models.observation import Observation

# Перечисление типов небесных тел
class BodyType(PyEnum):
    """Типы небесных тел"""
    PLANET = "planet"
    STAR = "star"
    GALAXY = "galaxy"
    NEBULA = "nebula"
    COMET = "comet"
    ASTEROID = "asteroid"
    BLACK_HOLE = "black_hole"

# Перечисление спектральных классов звезд
class SpectralClass(PyEnum):
    """
    Спектральные классы звезд (последовательность Гарвардская).

    Температурная шкала:
    - O: Самые горячие (>30000K)
    - B: Горячие (10000-30000K)
    - A: Белые (7500-10000K)
    - F: Желто-белые (6000-7500K)
    - G: Желтые (5200-6000K) - как наше Солнце
    - K: Оранжевые (3700-5200K)
    - M: Красные (<3700K)
    """
    O = "O"
```

```
B = "B"
A = "A"
F = "F"
G = "G"
K = "K"
M = "M"

class CelestialBodyBase(SQLModel):
    """
    Базовая модель небесного тела (для создания и обновления).
    """

    name: str = Field(
        ...,
        min_length=1,
        max_length=200,
        description="Название небесного тела",
        index=True
    )

    type: BodyType = Field(
        ...,
        description="Тип небесного тела",
        sa_column=Field(sa_column=SQLEnum(BodyType))
    )

    description: Optional[str] = Field(
        default=None,
        description="Описание небесного тела"
    )

    # ===== Физические характеристики =====

    mass: Optional[float] = Field(
        default=None,
        ge=0,
        description="Масса в массах Солнца (для звезд) или Земли
(для планет)"
    )
```

```
radius: Optional[float] = Field(
    default=None,
    ge=0,
    description="Радиус в радиусах Солнца или Земли"
)

temperature: Optional[float] = Field(
    default=None,
    ge=0,
    description="Температура поверхности в Кельвинах"
)

distance_from_earth: Optional[float] = Field(
    default=None,
    ge=0,
    description="Расстояние от Земли в световых годах"
)

# ===== Астрономические характеристики =====

spectral_class: Optional[SpectralClass] = Field(
    default=None,
    description="Спектральный класс (только для звезд)",
    sa_column=Field(sa_column=SQLEnum(SpectralClass))
)

absolute_magnitude: Optional[float] = Field(
    default=None,
    description="Абсолютная звёздная величина"
)

apparent_magnitude: Optional[float] = Field(
    default=None,
    description="Видимая звёздная величина"
)

# ===== Координаты в небесной сфере =====

right_ascension: Optional[float] = Field(
```

```
        default=None,
        ge=0,
        le=24,
        description="Прямое восхождение (часы, 0-24)"
    )

    declination: Optional[float] = Field(
        default=None,
        ge=-90,
        le=90,
        description="Склонение (градусы, -90 до 90)"
    )

# ===== Связи =====

parent_id: Optional[int] = Field(
    default=None,
    foreign_key="celestial_bodies.id",
    description="ID родительского небесного тела"
)
```

```
class CelestialBody(CelestialBodyBase, table=True):
```

```
    """
```

```
    Модель небесного тела для базы данных.
```

Структура:

- Основная информация (название, тип, описание)
 - Физические характеристики (масса, радиус, температура)
 - Астрономические характеристики (звездная величина, спектральный класс)
 - Координаты в небесной сфере
 - Связи с другими объектами (родитель, дети, наблюдения, наблюдатели)
- ```
 """
```

```
__tablename__ = "celestial_bodies"
```

```
Первичный ключ
id: int = Field(
```

```
 default=None,
 primary_key=True,
 index=True
)

 # Временные метки
 created_at: datetime = Field(
 default_factory=lambda: datetime.now(),
 nullable=False
)

 updated_at: datetime = Field(
 default_factory=lambda: datetime.now(),
 nullable=False
)

===== Связи с другими объектами =====

Связь с родительским телом
parent: Optional["CelestialBody"] = Relationship(
 back_populates="children",
 sa_relationship_kwargs={"remote_side":
"CelestialBody.id"}
)

Связь с дочерними телами
children: List["CelestialBody"] = Relationship(
 back_populates="parent"
)

Связь с наблюдениями
observations: List["Observation"] = Relationship(
 back_populates="celestial_body",
 sa_relationship_kwargs={"cascade": "all, delete-orphan"}
)

Индексы
__table_args__ = (
 Index("idx_type_distance", "type",
"distance_from_earth"),
```

```
 Index("idx_magnitude", "apparent_magnitude"),
)

class CelestialBodyCreate(CelestialBodyBase):
 """
 Схема для создания небесного тела.

 Наследует все поля от базовой модели.
 """
 pass

class CelestialBodyUpdate(SQLModel):
 """
 Схема для обновления небесного тела.

 Все поля опциональны, так как можно обновить только часть
 данных.
 """

 name: Optional[str] = Field(
 default=None,
 min_length=1,
 max_length=200
)

 type: Optional[BodyType] = None
 description: Optional[str] = None
 mass: Optional[float] = None
 radius: Optional[float] = None
 temperature: Optional[float] = None
 distance_from_earth: Optional[float] = None
 spectral_class: Optional[SpectralClass] = None
 absolute_magnitude: Optional[float] = None
 apparent_magnitude: Optional[float] = None
 right_ascension: Optional[float] = None
 declination: Optional[float] = None
 parent_id: Optional[int] = None
```

```
class CelestialBodyRead(CelestialBodyBase):
 """
 Схема для чтения небесного тела (полная информация).

 Используется для ответов из базы данных.
 """

 id: int
 created_at: datetime
 updated_at: datetime

 # Вычисляемые поля
 parent_name: Optional[str] = None
 children_count: int = 0
 observation_count: int = 0
 observers: Optional[List[dict]] = None
```

## app/models/astronomer.py

Содержание

### **astronomer.py**

---

```
"""
Модель астрономов.

Содержит информацию об ученых и их достижениях.
"""

from sqlmodel import SQLModel, Field, Relationship, Index
from typing import Optional, List, TYPE_CHECKING
from datetime import date, datetime

Импорты для связей
if TYPE_CHECKING:
 from app.models.observation import Observation
 from app.models.celestial_body import CelestialBody
```

```
class AstronomerBase(SQLModel):
 """
 Базовая модель астронома.
 """

 first_name: str = Field(
 ...,
 min_length=1,
 max_length=100,
 description="Имя астронома"
)

 last_name: str = Field(
 ...,
 min_length=1,
 max_length=100,
 description="Фамилия астронома"
)

 birth_date: Optional[date] = Field(
 default=None,
 description="Дата рождения"
)

 death_date: Optional[date] = Field(
 default=None,
 description="Дата смерти"
)

 nationality: Optional[str] = Field(
 default=None,
 max_length=100,
 description="Национальность"
)

 biography: Optional[str] = Field(
 default=None,
 description="Биография"
)
```

```
achievements: Optional[str] = Field(
 default=None,
 description="Научные достижения"
)

notable_discoveries: Optional[str] = Field(
 default=None,
 description="Известные открытия"
)

academic_degree: Optional[str] = Field(
 default=None,
 max_length=100,
 description="Академическая степень"
)

institution: Optional[str] = Field(
 default=None,
 max_length=200,
 description="Место работы"
)

is_active: bool = Field(
 default=True,
 description="Активен ли астроном (жив/работает)"
)

class Astronomer(AstronomerBase, table=True):
 """
 Модель астронома для базы данных.
 """

 __tablename__ = "astronomers"

 id: int = Field(
 default=None,
 primary_key=True,
 index=True
)
```

```
 created_at: datetime = Field(
 default_factory=lambda: datetime.now(),
 nullable=False
)

 updated_at: datetime = Field(
 default_factory=lambda: datetime.now(),
 nullable=False
)

 # Связь с наблюдениями
 observations: List["Observation"] = Relationship(
 back_populates="astronomer",
 sa_relationship_kwargs={"cascade": "all, delete-orphan"}
)

 __table_args__ = (
 Index("idx_astronomer_name", "first_name", "last_name"),
)
}

class AstronomerCreate(AstronomerBase):
 """Схема для создания астронома"""
 pass

class AstronomerUpdate(SQLModel):
 """Схема для обновления астронома"""

 first_name: Optional[str] = Field(default=None,
 min_length=1, max_length=100)
 last_name: Optional[str] = Field(default=None, min_length=1,
 max_length=100)
 birth_date: Optional[date] = None
 death_date: Optional[date] = None
 nationality: Optional[str] = None
 biography: Optional[str] = None
 achievements: Optional[str] = None
 notable_discoveries: Optional[str] = None
```

```
academic_degree: Optional[str] = None
institution: Optional[str] = None
is_active: Optional[bool] = None

class AstronomerRead(AstronomerBase):
 """Схема для чтения астронома"""

 id: int
 created_at: datetime
 updated_at: datetime

 # Вычисляемые поля
 observation_count: int = 0
 observed_bodies_count: int = 0
 observed_bodies: Optional[List[dict]] = None
```

## app/models/observation.py

### Содержание

#### **observation.py**

```
"""
Модель наблюдений астрономов за небесными телами.

Промежуточная таблица для связи многие-ко-многим.

"""

from sqlmodel import SQLModel, Field, Relationship, Index
from typing import Optional, TYPE_CHECKING
from datetime import datetime

Импорты для связей
if TYPE_CHECKING:
 from app.models.astronomer import Astronomer
 from app.models.celestial_body import CelestialBody

class ObservationBase(SQLModel):
```

```
"""
Базовая модель наблюдения.

"""

astronomer_id: int = Field(
 ...,
 ge=1,
 foreign_key="astronomers.id",
 description="ID астронома"
)

celestial_body_id: int = Field(
 ...,
 ge=1,
 foreign_key="celestial_bodies.id",
 description="ID небесного тела"
)

observation_date: datetime = Field(
 ...,
 description="Дата и время наблюдения"
)

location: Optional[str] = Field(
 default=None,
 max_length=200,
 description="Место наблюдения (обсерватория)"
)

equipment: Optional[str] = Field(
 default=None,
 max_length=200,
 description="Используемое оборудование"
)

duration_hours: Optional[float] = Field(
 default=None,
 ge=0,
 description="Продолжительность наблюдения в часах"
)
```

```
 weather_conditions: Optional[str] = Field(
 default=None,
 max_length=100,
 description="Погодные условия"
)

 notes: Optional[str] = Field(
 default=None,
 description="Заметки и комментарии"
)

 data_collected: Optional[str] = Field(
 default=None,
 description="Научные данные"
)
```

```
class Observation(ObservationBase, table=True):
```

```
 """
```

```
 Модель наблюдения для базы данных.
```

```
 Связывает астронома с небесным телом и содержит детали
наблюдения.
```

```
 Это промежуточная таблица для связи многие-ко-многим.
```

```
 """
```

```
 __tablename__ = "observations"
```

```
 id: int = Field(
 default=None,
 primary_key=True,
 index=True
)
```

```
 created_at: datetime = Field(
 default_factory=lambda: datetime.now(),
 nullable=False
)
```

```
updated_at: datetime = Field(
 default_factory=lambda: datetime.now(),
 nullable=False
)

Связи
astronomer: "Astronomer" =
Relationship(back_populates="observations")
celestial_body: "CelestialBody" =
Relationship(back_populates="observations")

__table_args__ = (
 Index("idx_observation_date", "observation_date"),
 Index("idx_astronomer_celestial", "astronomer_id",
"celestial_body_id"),
)
```

```
class ObservationCreate(ObservationBase):
 """Схема для создания наблюдения"""
 pass
```

```
class ObservationUpdate(SQLModel):
 """Схема для обновления наблюдения"""

 location: Optional[str] = Field(default=None,
max_length=200)
 equipment: Optional[str] = Field(default=None,
max_length=200)
 duration_hours: Optional[float] = Field(default=None, ge=0)
 weather_conditions: Optional[str] = Field(default=None,
max_length=100)
 notes: Optional[str] = None
 data_collected: Optional[str] = None
```

```
class ObservationRead(ObservationBase):
 """Схема для чтения наблюдения"""
```

```
 id: int
 created_at: datetime
 updated_at: datetime

 # Дополнительная информация
 astronomer_name: Optional[str] = None
 celestial_body_name: Optional[str] = None
```

## app/models/user.py

Содержание

### user.py

```
"""
Модель пользователей для аутентификации.
"""

from sqlmodel import SQLModel, Field
from typing import Optional
from datetime import datetime
from pydantic import EmailStr, field_validator
```

```
class UserBase(SQLModel):
 """
 Базовая модель пользователя.

 username: str = Field(
 ...,
 min_length=3,
 max_length=50,
 description="Имя пользователя",
 unique=True,
 index=True
)

 email: EmailStr = Field(
 ...,
 description="Email пользователя",
 unique=True
)
```

```
 unique=True
)

 full_name: Optional[str] = Field(
 default=None,
 max_length=100,
 description="Полное имя"
)

class User(UserBase, table=True):
 """
 Модель пользователя для базы данных.
 """

 __tablename__ = "users"

 id: int = Field(
 default=None,
 primary_key=True,
 index=True
)

 hashed_password: str = Field(
 ...,
 description="Хешированный пароль"
)

 is_active: bool = Field(
 default=True,
 description="Активен ли пользователь"
)

 created_at: datetime = Field(
 default_factory=lambda: datetime.now(),
 nullable=False
)

 updated_at: datetime = Field(
 default_factory=lambda: datetime.now(),
```

```
 nullable=False
)

class UserCreate(UserBase):
 """
 Схема для создания пользователя.
 """

 password: str = Field(
 ...,
 min_length=8,
 description="Пароль (минимум 8 символов)"
)

 @field_validator("password")
 @classmethod
 def validate_password(cls, v):
 """Валидация пароля"""
 if len(v) < 8:
 raise ValueError("Пароль должен быть минимум 8 символов")
 if not any(c.isupper() for c in v):
 raise ValueError("Пароль должен содержать хотя бы одну заглавную букву")
 if not any(c.isdigit() for c in v):
 raise ValueError("Пароль должен содержать хотя бы одну цифру")
 return v

class UserUpdate(SQLModel):
 """
 Схема для обновления пользователя.
 """

 email: Optional[EmailStr] = None
 full_name: Optional[str] = Field(default=None,
 max_length=100)
 password: Optional[str] = Field(default=None, min_length=8)
```

```
class UserRead(UserBase):
 """
 Схема для чтения пользователя.
 """

 id: int
 is_active: bool
 created_at: datetime
 updated_at: datetime

class UserLogin(SQLModel):
 """
 Схема для входа в систему.
 """

 username: str = Field(..., description="Имя пользователя или email")
 password: str = Field(..., description="Пароль")
```

## app/routers/init.py

Содержание

### \_\_init\_\_.py

```
"""
Маршруты (routers) приложения.

Содержат все эндпоинты API.
"""

from app.routers.celestial_bodies import router as celestial_bodies_router
from app.routers.astronomers import router as astronomers_router
from app.routers.observations import router as observations_router
from app.routers.auth import router as auth_router
```

```
all = [
 "celestial_bodies_router",
 "astronomers_router",
 "observations_router",
 "auth_router"
]
```

## app/routers/celestial\_bodies.py

### Содержание

#### celestial\_bodies.py

```
"""
Маршрут для работы с небесными телами.

Содержит все CRUD операции и дополнительные методы.
"""

from fastapi import APIRouter, Depends, HTTPException, Query,
status
from sqlmodel.ext.asyncio.session import AsyncSession
from sqlmodel import select, func
from typing import List, Optional
from datetime import datetime

from app.database import get_db
from app.models.celestial_body import (
 CelestialBody,
 BodyType,
 SpectralClass,
 CelestialBodyCreate,
 CelestialBodyUpdate,
 CelestialBodyRead
)
from app.services.search import apply_celestial_body_filters

router = APIRouter(
 prefix="/celestial-bodies",
 tags=["Небесные тела"],
```

```
 responses={404: {"description": "Не найдено"}}
)

===== CRUD операции =====

@router.post(
 "/",
 response_model=CelestialBodyRead,
 status_code=status.HTTP_201_CREATED,
 summary="Создать небесное тело",
 description="Создает новую запись о небесном теле в базе
данных"
)
async def create_celestial_body(
 body: CelestialBodyCreate,
 db: AsyncSession = Depends(get_db)
):
 """
 Создание нового небесного тела.

 Параметры:
 - `body`: данные о небесном теле

 Возвращает:
 - Созданное небесное тело с полной информацией
 """

 # Проверка существования тела с таким именем
 query = select(CelestialBody).where(CelestialBody.name ==
body.name)
 result = await db.execute(query)
 existing = result.scalar_one_or_none()

 if existing:
 raise HTTPException(
 status_code=status.HTTP_400_BAD_REQUEST,
 detail=f"Небесное тело с именем '{body.name}' уже
существует"
)
```

```
Проверка существования родительского тела
if body.parent_id:
 parent_query =
select(CelestialBody).where(CelestialBody.id == body.parent_id)
 parent_result = await db.execute(parent_query)
 parent = parent_result.scalar_one_or_none()

 if not parent:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Родительское тело с ID {body.parent_id} не найдено"
)

Создание нового объекта
db_body = CelestialBody(**body.model_dump())

Добавление в сессию
db.add(db_body)

Фиксация изменений
await db.commit()

Обновление объекта
await db.refresh(db_body)

Создание ответа с вычисляемыми полями
response = CelestialBodyRead(**db_body.model_dump())
response.parent_name = db_body.parent.name if db_body.parent
else None
response.children_count = len(db_body.children)
response.observation_count = len(db_body.observations)
response.observers = [
 {"id": obs.astronomer.id, "name":
 obs.astronomer.first_name + " " + obs.astronomer.last_name}
 for obs in db_body.observations
] if db_body.observations else []

return response
```

```
@router.get(
 "/",
 response_model=List[CelestialBodyRead],
 summary="Получить список небесных тел",
 description="Возвращает список небесных тел с пагинацией и
фильтрацией"
)
async def read_celestial_bodies(
 skip: int = Query(0, ge=0, description="Количество
пропущенных записей"),
 limit: int = Query(10, ge=1, le=100, description="Количество
записей"),
 search: Optional[str] = Query(None, description="Поиск по
названию"),
 body_type: Optional[BodyType] = Query(None,
description="Фильтр по типу"),
 db: AsyncSession = Depends(get_db)
):
 """

```

Получение списка небесных тел с возможностью фильтрации.

**\*\*Параметры:\*\***

- `skip`: количество пропущенных записей (для пагинации)
- `limit`: количество записей на страницу
- `search`: поиск по названию
- `body\_type`: фильтр по типу тела

**\*\*Возвращает:\*\***

- Список небесных тел

"""

```
Создание базового запроса
query = select(CelestialBody)
```

```
Применение фильтров
```

```
if search:
```

```
 query =
```

```
query.where(CelestialBody.name.ilike(f"%{search}%"))
```

```
if body_type:
 query = query.where(CelestialBody.type == body_type)

Применение пагинации
query = query.offset(skip).limit(limit)

Выполнение запроса
result = await db.execute(query)
bodies = result.scalars().all()

Преобразование в схемы с вычисляемыми полями
response_list = []
for body in bodies:
 body_read = CelestialBodyRead(**body.model_dump())
 body_read.parent_name = body.parent.name if body.parent
else None
 body_read.children_count = len(body.children)
 body_read.observation_count = len(body.observations)
 body_read.observers = [
 {"id": obs.astronomer.id, "name": f'{obs.astronomer.first_name} {obs.astronomer.last_name}'}
 for obs in body.observations
] if body.observations else []
 response_list.append(body_read)

return response_list

@router.get(
 "/{body_id}",
 response_model=CelestialBodyRead,
 summary="Получить небесное тело по ID",
 description="Возвращает подробную информацию о небесном
 теле"
)
async def read_celestial_body(
 body_id: int,
 db: AsyncSession = Depends(get_db)
):
```

```
"""
Получение небесного тела по ID.

Параметры:
- `body_id`: ID небесного тела

Возвращает:
- Подробная информация о небесном теле
"""

Получение тела по ID
query = select(CelestialBody).where(CelestialBody.id == body_id)
result = await db.execute(query)
body = result.scalar_one_or_none()

if not body:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Небесное тело с ID {body_id} не найдено"
)

Создание ответа с вычисляемыми полями
response = CelestialBodyRead(**body.model_dump())
response.parent_name = body.parent.name if body.parent else None
response.children_count = len(body.children)
response.observation_count = len(body.observations)
response.observers = [
 {"id": obs.astronomer.id, "name": f"{obs.astronomer.first_name} {obs.astronomer.last_name}"}
 for obs in body.observations
] if body.observations else []

return response

@router.put(
 "/{body_id}",
 response_model=CelestialBodyRead,
```

```
summary="Обновить небесное тело",
description="Обновляет информацию о небесном теле"
)
async def update_celestial_body(
 body_id: int,
 body_update: CelestialBodyUpdate,
 db: AsyncSession = Depends(get_db)
):
 """
 Обновление небесного тела.

 Параметры:
 - `body_id`: ID небесного тела
 - `body_update`: данные для обновления

 Возвращает:
 - Обновленное небесное тело
 """

Получение существующего тела
query = select(CelestialBody).where(CelestialBody.id == body_id)
result = await db.execute(query)
db_body = result.scalar_one_or_none()

if not db_body:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Небесное тело с ID {body_id} не найдено"
)

Обновление полей
update_data = body_update.model_dump(exclude_unset=True)

for field, value in update_data.items():
 setattr(db_body, field, value)

await db.commit()
await db.refresh(db_body)
```

```
Создание ответа
response = CelestialBodyRead(**db_body.model_dump())
response.parent_name = db_body.parent.name if db_body.parent
else None
response.children_count = len(db_body.children)
response.observation_count = len(db_body.observations)
response.observers = [
 {"id": obs.astronomer.id, "name": f'{obs.astronomer.first_name} {obs.astronomer.last_name}'}
 for obs in db_body.observations
] if db_body.observations else []

return response
```

```
@router.delete(
 "/{body_id}",
 status_code=status.HTTP_204_NO_CONTENT,
 summary="Удалить небесное тело",
 description="Удаляет небесное тело из базы данных"
)
```

```
async def delete_celestial_body(
 body_id: int,
 db: AsyncSession = Depends(get_db)
):
```

```
 """
 Удаление небесного тела.
```

```
 Параметры:
```

```
- `body_id`: ID небесного тела
```

```
 Возвращает:
```

```
- 204 No Content при успешном удалении
```

```
 """
```

```
 query = select(CelestialBody).where(CelestialBody.id ==
body_id)
 result = await db.execute(query)
 db_body = result.scalar_one_or_none()
```

```
 if not db_body:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Небесное тело с ID {body_id} не найдено"
)

 # Удаление объекта
 await db.delete(db_body)
 await db.commit()

 return None

====== Расширенные методы ======
@router.get(
 "/statistics",
 summary="Статистика по небесным телам",
 description="Возвращает статистику по типам небесных тел"
)
async def get_statistics(db: AsyncSession = Depends(get_db)):
 """
 Получение статистики по небесным телам.

 Возвращает:
 - Количество тел по типам
 - Общее количество
 - Статистика по расстояниям
 """

 # Подсчет количества по типам
 query = (
 select(CelestialBody.type, func.count(CelestialBody.id))
 .group_by(CelestialBody.type)
)

 result = await db.execute(query)
 type_counts = {type_.value: count for type_, count in
result.all()}


```

```
Общее количество
total_query = select(func.count(CelestialBody.id))
total_result = await db.execute(total_query)
total = total_result.scalar()

Статистика по расстояниям
distance_query = select(
 func.avg(CelestialBody.distance_from_earth),
 func.min(CelestialBody.distance_from_earth),
 func.max(CelestialBody.distance_from_earth)
).where(CelestialBody.distance_from_earth.isnot(None))

distance_result = await db.execute(distance_query)
avg_dist, min_dist, max_dist = distance_result.first()

return {
 "total": total,
 "by_type": type_counts,
 "distance_statistics": {
 "average": avg_dist,
 "minimum": min_dist,
 "maximum": max_dist
 }
}
```

## app/routers/astronomers.py

Содержание

### astronomers.py

```
"""
Маршрут для работы с астрономами.
"""

from fastapi import APIRouter, Depends, HTTPException, Query, status
from sqlmodel.ext.asyncio.session import AsyncSession
from sqlmodel import select, func
from typing import List, Optional
from datetime import datetime
```

```
from app.database import get_db
from app.models.astronomer import (
 Astronomer,
 AstronomerCreate,
 AstronomerUpdate,
 AstronomerRead
)

router = APIRouter(
 prefix="/astronomers",
 tags=["Астрономы"],
 responses={404: {"description": "Не найдено"}}
)

@router.post(
 "/",
 response_model=AstronomerRead,
 status_code=status.HTTP_201_CREATED,
 summary="Создать астронома",
 description="Создает новую запись об астрономе"
)
async def create_astronomer(
 astronomer: AstronomerCreate,
 db: AsyncSession = Depends(get_db)
):
 """Создание нового астронома"""

 # Проверка существования
 query = select(Astronomer).where(
 (Astronomer.first_name == astronomer.first_name) &
 (Astronomer.last_name == astronomer.last_name)
)
 result = await db.execute(query)
 existing = result.scalar_one_or_none()

 if existing:
 raise HTTPException(
```

```
 status_code=status.HTTP_400_BAD_REQUEST,
 detail=f"Астроном {astronomer.first_name}"
{astronomer.last_name} уже существует"
)

db_astronomer = Astronomer(**astronomer.model_dump())
db.add(db_astronomer)
await db.commit()
await db.refresh(db_astronomer)

Создание ответа с вычисляемыми полями
response = AstronomerRead(**db_astronomer.model_dump())
response.observation_count = len(db_astronomer.observations)
response.observed_bodies_count = len(set(
 obs.celestial_body_id for obs in
db_astronomer.observations
)) if db_astronomer.observations else 0
response.observed_bodies = [
{
 "id": obs.celestial_body.id,
 "name": obs.celestial_body.name,
 "type": obs.celestial_body.type.value
}
 for obs in db_astronomer.observations
] if db_astronomer.observations else []

return response

@router.get(
 "/",
 response_model=List[AstronomerRead],
 summary="Получить список астрономов"
)
async def read_astronomers(
 skip: int = Query(0, ge=0),
 limit: int = Query(10, ge=1, le=100),
 search: Optional[str] = Query(None),
 is_active: Optional[bool] = Query(None),
 db: AsyncSession = Depends(get_db)
```

```
):
 """Получение списка астрономов"""

 query = select(Astronomer)

 if search:
 query = query.where(
 (Astronomer.first_name.ilike(search)) |
 (Astronomer.last_name.ilike(search))
)

 if is_active is not None:
 query = query.where(Astronomer.is_active == is_active)

 query = query.offset(skip).limit(limit)

 result = await db.execute(query)
 astronomers = result.scalars().all()

 # Преобразование в схемы с вычисляемыми полями
 response_list = []
 for astronomer in astronomers:
 astro_read = AstronomerRead(**astronomer.model_dump())
 astro_read.observation_count =
 len(astronomer.observations)
 astro_read.observed_bodies_count = len(set(
 obs.celestial_body_id for obs in
 astronomer.observations
)) if astronomer.observations else 0
 astro_read.observed_bodies = [
 {
 "id": obs.celestial_body.id,
 "name": obs.celestial_body.name,
 "type": obs.celestial_body.type.value
 }
 for obs in astronomer.observations
] if astronomer.observations else []
 response_list.append(astro_read)

 return response_list
```

```
@router.get(
 "/{astronomer_id}",
 response_model=AstronomerRead,
 summary="Получить астронома по ID"
)
async def read_astronomer(
 astronomer_id: int,
 db: AsyncSession = Depends(get_db)
):
 """Получение астронома по ID"""

 query = select(Astronomer).where(Astronomer.id ==
astronomer_id)
 result = await db.execute(query)
 astronomer = result.scalar_one_or_none()

 if not astronomer:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Астроном с ID {astronomer_id} не найден"
)

 # Создание ответа с вычисляемыми полями
 response = AstronomerRead(**astronomer.model_dump())
 response.observation_count = len(astronomer.observations)
 response.observed_bodies_count = len(set(
 obs.celestial_body_id for obs in astronomer.observations
)) if astronomer.observations else 0
 response.observed_bodies = [
 {
 "id": obs.celestial_body.id,
 "name": obs.celestial_body.name,
 "type": obs.celestial_body.type.value
 }
 for obs in astronomer.observations
] if astronomer.observations else []

 return response
```

```
@router.put(
 "/{astronomer_id}",
 response_model=AstronomerRead,
 summary="Обновить астронома"
)
async def update_astronomer(
 astronomer_id: int,
 astronomer_update: AstronomerUpdate,
 db: AsyncSession = Depends(get_db)
):
 """Обновление астронома"""

 query = select(Astronomer).where(Astronomer.id == astronomer_id)
 result = await db.execute(query)
 db_astronomer = result.scalar_one_or_none()

 if not db_astronomer:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Астроном с ID {astronomer_id} не найден"
)

 update_data =
 astronomer_update.model_dump(exclude_unset=True)

 for field, value in update_data.items():
 setattr(db_astronomer, field, value)

 await db.commit()
 await db.refresh(db_astronomer)

 # Создание ответа
 response = AstronomerRead(**db_astronomer.model_dump())
 response.observation_count = len(db_astronomer.observations)
 response.observed_bodies_count = len(set(
 obs.celestial_body_id for obs in
 db_astronomer.observations
```

```
)> if db_astronomer.observations else 0
response.observed_bodies = [
 {
 "id": obs.celestial_body.id,
 "name": obs.celestial_body.name,
 "type": obs.celestial_body.type.value
 }
 for obs in db_astronomer.observations
] if db_astronomer.observations else []

return response

@router.delete(
 "/{astronomer_id}",
 status_code=status.HTTP_204_NO_CONTENT,
 summary="Удалить астронома"
)
async def delete_astronomer(
 astronomer_id: int,
 db: AsyncSession = Depends(get_db)
):
 """Удаление астронома"""

 query = select(Astronomer).where(Astronomer.id ==
astronomer_id)
 result = await db.execute(query)
 db_astronomer = result.scalar_one_or_none()

 if not db_astronomer:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Астроном с ID {astronomer_id} не найден"
)

 await db.delete(db_astronomer)
 await db.commit()

return None
```

```
@router.get(
 "/statistics",
 summary="Статистика по астрономам"
)
async def get_astronomer_statistics(db: AsyncSession =
Depends(get_db)):
 """Получение статистики по астрономам"""

 # Общее количество
 total_query = select(func.count(Astronomer.id))
 total_result = await db.execute(total_query)
 total = total_result.scalar()

 # Количество активных
 active_query =
select(func.count(Astronomer.id)).where(Astronomer.is_active ==
True)
 active_result = await db.execute(active_query)
 active = active_result.scalar()

 # Статистика по национальностям
 nationality_query = (
 select(Astronomer.nationality,
func.count(Astronomer.id))
 .where(Astronomer.nationality.isnot(None))
 .group_by(Astronomer.nationality)
 .order_by(func.count(Astronomer.id).desc())
)
 nationality_result = await db.execute(nationality_query)
 nationalities = {nat: count for nat, count in
nationality_result.all()}

 return {
 "total": total,
 "active": active,
 "inactive": total - active,
 "by_nationality": nationalities
 }
```

## app/routers/observations.py

### Содержание

#### observations.py

```
"""
Маршрут для работы с наблюдениями.

"""

from fastapi import APIRouter, Depends, HTTPException, Query, status
from sqlmodel.ext.asyncio.session import AsyncSession
from sqlmodel import select, func
from typing import List, Optional
from datetime import datetime

from app.database import get_db
from app.models.observation import (
 Observation,
 ObservationCreate,
 ObservationUpdate,
 ObservationRead
)
from app.models.astronomer import Astronomer
from app.models.celestial_body import CelestialBody

router = APIRouter(
 prefix="/observations",
 tags=["Наблюдения"],
 responses={404: {"description": "Не найдено"}}
)

@router.post(
 "/",
 response_model=ObservationRead,
 status_code=status.HTTP_201_CREATED,
 summary="Создать наблюдение"
)
async def create_observation(
```

```
 observation: ObservationCreate,
 db: AsyncSession = Depends(get_db)
):
 """Создание нового наблюдения"""

 # Проверка существования астронома
 astronomer_query = select(Astronomer).where(Astronomer.id == observation.astronomer_id)
 astronomer_result = await db.execute(asntronomer_query)
 astronomer = astronomer_result.scalar_one_or_none()

 if not astronomer:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Астроном с ID {observation.astronomer_id} не найден"
)

 # Проверка существования небесного тела
 body_query = select(CelestialBody).where(CelestialBody.id == observation.celestial_body_id)
 body_result = await db.execute(body_query)
 celestial_body = body_result.scalar_one_or_none()

 if not celestial_body:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Небесное тело с ID {observation.celestial_body_id} не найдено"
)

 # Проверка дубликата наблюдения
 duplicate_query = select(Observation).where(
 (Observation.astronomer_id == observation.astronomer_id) &
 (Observation.celestial_body_id == observation.celestial_body_id) &
 (func.date(Observation.observation_date) == func.date(observation.observation_date))
)
```

```
 duplicate_result = await db.execute(duplicate_query)
 duplicate = duplicate_result.scalar_one_or_none()

 if duplicate:
 raise HTTPException(
 status_code=status.HTTP_400_BAD_REQUEST,
 detail="Астроном уже проводил наблюдение этого тела
в эту дату"
)

 db_observation = Observation(**observation.model_dump())
 db.add(db_observation)
 await db.commit()
 await db.refresh(db_observation)

 # Создание ответа с дополнительной информацией
 response = ObservationRead(**db_observation.model_dump())
 response.astronomer_name = f"{astronomer.first_name} {astronomer.last_name}"
 response.celestial_body_name = celestial_body.name

 return response

@router.get(
 "/",
 response_model=List[ObservationRead],
 summary="Получить список наблюдений"
)
async def read_observations(
 skip: int = Query(0, ge=0),
 limit: int = Query(10, ge=1, le=100),
 astronomer_id: Optional[int] = Query(None),
 celestial_body_id: Optional[int] = Query(None),
 db: AsyncSession = Depends(get_db)
):
 """Получение списка наблюдений с фильтрацией"""

 query = select(Observation)
```

```
if astronomer_id:
 query = query.where(Observation.astronomer_id ==
astronomer_id)

if celestial_body_id:
 query = query.where(Observation.celestial_body_id ==
celestial_body_id)

query = query.order_by(Observation.observation_date.desc())
query = query.offset(skip).limit(limit)

result = await db.execute(query)
observations = result.scalars().all()

Преобразование в схемы с дополнительной информацией
response_list = []
for obs in observations:
 obs_read = ObservationRead(**obs.model_dump())

 # Получение связанных данных
 astro_query = select(Astronomer).where(Astronomer.id ==
obs.astronomer_id)
 astro_result = await db.execute(astro_query)
 astronomer = astro_result.scalar_one_or_none()

 body_query =
select(CelestialBody).where(CelestialBody.id ==
obs.celestial_body_id)
 body_result = await db.execute(body_query)
 celestial_body = body_result.scalar_one_or_none()

 obs_read.astronomer_name = f"{astronomer.first_name} {astronomer.last_name}" if astronomer else None
 obs_read.celestial_body_name = celestial_body.name if celestial_body else None

 response_list.append(obs_read)

return response_list
```

```
@router.get(
 "/{observation_id}",
 response_model=ObservationRead,
 summary="Получить наблюдение по ID"
)
async def read_observation(
 observation_id: int,
 db: AsyncSession = Depends(get_db)
):
 """Получение наблюдения по ID"""

 query = select(Observation).where(Observation.id ==
observation_id)
 result = await db.execute(query)
 observation = result.scalar_one_or_none()

 if not observation:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Наблюдение с ID {observation_id} не
найдено"
)

 # Создание ответа с дополнительной информацией
 response = ObservationRead(**observation.model_dump())

 astro_query = select(Astronomer).where(Astronomer.id ==
observation.astronomer_id)
 astro_result = await db.execute(astro_query)
 astronomer = astro_result.scalar_one_or_none()

 body_query = select(CelestialBody).where(CelestialBody.id ==
observation.celestial_body_id)
 body_result = await db.execute(body_query)
 celestial_body = body_result.scalar_one_or_none()

 response.astronomer_name = f"{astronomer.first_name} {astronomer.last_name}" if astronomer else None
 response.celestial_body_name = celestial_body.name if
```

```
celestial_body else None

return response

@router.put(
 "/{observation_id}",
 response_model=ObservationRead,
 summary="Обновить наблюдение"
)
async def update_observation(
 observation_id: int,
 observation_update: ObservationUpdate,
 db: AsyncSession = Depends(get_db)
):
 """Обновление наблюдения"""

 query = select(Observation).where(Observation.id ==
observation_id)
 result = await db.execute(query)
 db_observation = result.scalar_one_or_none()

 if not db_observation:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Наблюдение с ID {observation_id} не
найдено"
)

 update_data =
observation_update.model_dump(exclude_unset=True)

 for field, value in update_data.items():
 setattr(db_observation, field, value)

 await db.commit()
 await db.refresh(db_observation)

 # Создание ответа
 response = ObservationRead(**db_observation.model_dump())
```

```
 astro_query = select(Astronomer).where(Astronomer.id == db_observation.astronomer_id)
 astro_result = await db.execute(astro_query)
 astronomer = astro_result.scalar_one_or_none()

 body_query = select(CelestialBody).where(CelestialBody.id == db_observation.celestial_body_id)
 body_result = await db.execute(body_query)
 celestial_body = body_result.scalar_one_or_none()

 response.astronomer_name = f"{astronomer.first_name}{astronomer.last_name}" if astronomer else None
 response.celestial_body_name = celestial_body.name if celestial_body else None

 return response

@router.delete(
 "/{observation_id}",
 status_code=status.HTTP_204_NO_CONTENT,
 summary="Удалить наблюдение"
)
async def delete_observation(
 observation_id: int,
 db: AsyncSession = Depends(get_db)
):
 """Удаление наблюдения"""

 query = select(Observation).where(Observation.id == observation_id)
 result = await db.execute(query)
 db_observation = result.scalar_one_or_none()

 if not db_observation:
 raise HTTPException(
 status_code=status.HTTP_404_NOT_FOUND,
 detail=f"Наблюдение с ID {observation_id} не найдено"
```

```
)

 await db.delete(db_observation)
 await db.commit()

 return None

@router.get(
 "/statistics",
 summary="Статистика по наблюдениям"
)
async def get_observation_statistics(db: AsyncSession = Depends(get_db)):
 """Получение статистики по наблюдениям"""

 # Общее количество наблюдений
 total_query = select(func.count(Observation.id))
 total_result = await db.execute(total_query)
 total = total_result.scalar()

 # Топ-5 астрономов по количеству наблюдений
 top_astronomers_query = (
 select(
 Astronomer.id,
 Astronomer.first_name,
 Astronomer.last_name,
 func.count(Observation.id).label('count')
)
 .join(Observation)
 .group_by(Astronomer.id, Astronomer.first_name,
 Astronomer.last_name)
 .order_by(func.count(Observation.id).desc())
 .limit(5)
)
 top_astronomers_result = await
 db.execute(top_astronomers_query)
 top_astronomers = [
 {
 "id": id_,
```

```
 "name": f"{first_name} {last_name}",
 "observation_count": count
 }
 for id_, first_name, last_name, count in
top_astronomers_result.all()
]

Топ-5 небесных тел по количеству наблюдений
top_bodies_query = (
 select(
 CelestialBody.id,
 CelestialBody.name,
 func.count(Observation.id).label('count')
)
 .join(Observation)
 .group_by(CelestialBody.id, CelestialBody.name)
 .order_by(func.count(Observation.id).desc())
 .limit(5)
)
top_bodies_result = await db.execute(top_bodies_query)
top_bodies = [
 {"id": id_, "name": name, "observation_count": count}
 for id_, name, count in top_bodies_result.all()
]

return {
 "total_observations": total,
 "top_astronomers": top_astronomers,
 "top_celestial_bodies": top_bodies
}
```

## app/routers/auth.py

Содержание

### auth.py

```
"""
```

Маршрут для аутентификации.

Включает регистрацию, вход и работу с токенами JWT.

```
"""

from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer,
OAuth2PasswordRequestForm
from sqlmodel.ext.asyncio.session import AsyncSession
from sqlmodel import select
from datetime import datetime, timedelta
from typing import Optional
from jose import JWTError, jwt
from passlib.context import CryptContext

from app.database import get_db
from app.models.user import (
 User,
 UserCreate,
 UserRead
)

Настройки для безопасности
from dotenv import load_dotenv
import os

load_dotenv()

SECRET_KEY = os.getenv("SECRET_KEY", "your-secret-key-change-in-production")
ALGORITHM = os.getenv("ALGORITHM", "HS256")
ACCESS_TOKEN_EXPIRE_MINUTES =
int(os.getenv("ACCESS_TOKEN_EXPIRE_MINUTES", "30"))

Настройки для хеширования паролей
pwd_context = CryptContext(schemes=["bcrypt"],
deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="auth/token")

router = APIRouter(
 prefix="/auth",
 tags=["Аутентификация"],
```

```
 responses={404: {"description": "Не найдено"}}
)

===== Вспомогательные функции =====

def verify_password(plain_password: str, hashed_password: str) \
> bool:
 """Проверка пароля"""
 return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password: str) → str:
 """Хеширование пароля"""
 return pwd_context.hash(password)

def create_access_token(data: dict, expires_delta: Optional[timedelta] = None) → str:
 """Создание JWT токена"""
 to_encode = data.copy()

 if expires_delta:
 expire = datetime.utcnow() + expires_delta
 else:
 expire = datetime.utcnow() +
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)

 to_encode.update({"exp": expire})
 encoded_jwt = jwt.encode(to_encode, SECRET_KEY,
algorithm=ALGORITHM)

 return encoded_jwt

async def get_current_user(
 token: str = Depends(oauth2_scheme),
 db: AsyncSession = Depends(get_db)
):
 """Получение текущего пользователя из токена"""

```

```
 credentials_exception = HTTPException(
 status_code=status.HTTP_401_UNAUTHORIZED,
 detail="Не удалось проверить учетные данные",
 headers={"WWW-Authenticate": "Bearer"},
)

 try:
 payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
 username: str = payload.get("sub")
 if username is None:
 raise credentials_exception
 except JWTError:
 raise credentials_exception

 # Получение пользователя из БД
 query = select(User).where(User.username == username)
 result = await db.execute(query)
 user = result.scalar_one_or_none()

 if user is None:
 raise credentials_exception

 return user

===== Эндпоинты =====

@router.post(
 "/register",
 response_model=UserRead,
 status_code=status.HTTP_201_CREATED,
 summary="Регистрация пользователя"
)
async def register_user(
 user: UserCreate,
 db: AsyncSession = Depends(get_db)
):
 """Регистрация нового пользователя"""

```

```
Проверка существования пользователя
query = select(User).where(
 (User.username == user.username) | (User.email ==
user.email)
)
result = await db.execute(query)
existing_user = result.scalar_one_or_none()

if existing_user:
 raise HTTPException(
 status_code=status.HTTP_400_BAD_REQUEST,
 detail="Пользователь с таким именем или email уже
существует"
)

Создание нового пользователя
hashed_password = get_password_hash(user.password)

db_user = User(
 username=user.username,
 email=user.email,
 full_name=user.full_name,
 hashed_password=hashed_password,
 is_active=True
)

db.add(db_user)
await db.commit()
await db.refresh(db_user)

Создание ответа без пароля
return UserRead(
 id=db_user.id,
 username=db_user.username,
 email=db_user.email,
 full_name=db_user.full_name,
 is_active=db_user.is_active,
 created_at=db_user.created_at,
 updated_at=db_user.updated_at
)
```

```
@router.post(
 "/token",
 summary="Получить токен доступа"
)
async def login_for_access_token(
 form_data: OAuth2PasswordRequestForm = Depends(),
 db: AsyncSession = Depends(get_db)
):
 """Получение токена доступа по логину и паролю"""

 # Получение пользователя из БД
 query = select(User).where(User.username ==
form_data.username)
 result = await db.execute(query)
 user = result.scalar_one_or_none()

 # Проверка пользователя
 if not user or not verify_password(form_data.password,
user.hashed_password):
 raise HTTPException(
 status_code=status.HTTP_401_UNAUTHORIZED,
 detail="Неверное имя пользователя или пароль",
 headers={"WWW-Authenticate": "Bearer"},
)

 # Проверка активности пользователя
 if not user.is_active:
 raise HTTPException(
 status_code=status.HTTP_400_BAD_REQUEST,
 detail="Пользователь неактивен"
)

 # Создание токена
 access_token = create_access_token(
 data={"sub": user.username},

 expires_delta=timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
)
```

```
 return {
 "access_token": access_token,
 "token_type": "bearer",
 "user": {
 "id": user.id,
 "username": user.username,
 "email": user.email
 }
 }

@router.get(
 "/me",
 response_model=UserRead,
 summary="Получить информацию о текущем пользователе"
)
async def read_users_me(
 current_user: User = Depends(get_current_user)
):
 """Получение информации о текущем пользователе"""

 return UserRead(
 id=current_user.id,
 username=current_user.username,
 email=current_user.email,
 full_name=current_user.full_name,
 is_active=current_user.is_active,
 created_at=current_user.created_at,
 updated_at=current_user.updated_at
)
```

## app/services/init.py

Содержание

### \_\_init\_\_.py

```
"""
```

Сервисы приложения.

Содержат бизнес-логику и вспомогательные функции.

....

## app/services/search.py

Содержание

### search.py

....

Сервис для расширенного поиска и фильтрации.

....

```
from sqlmodel import select
from sqlmodel.sql.expression import Select
from typing import Optional
from datetime import datetime

from app.models.celestial_body import CelestialBody, BodyType,
SpectralClass
```

```
def apply_celestial_body_filters(
 query: Select,
 name: Optional[str] = None,
 body_type: Optional[BodyType] = None,
 min_distance: Optional[float] = None,
 max_distance: Optional[float] = None,
 min_magnitude: Optional[float] = None,
 max_magnitude: Optional[float] = None,
 spectral_class: Optional[SpectralClass] = None
) → Select:
 """
```

Применяет фильтры поиска к запросу небесных тел.

**\*\*Параметры:\*\***

- `query`: базовый запрос
- `name`: поиск по названию
- `body\_type`: фильтр по типу
- `min\_distance`, `max\_distance`: фильтр по расстоянию
- `min\_magnitude`, `max\_magnitude`: фильтр по звёздной

величине

- `spectral\_class`: фильтр по спектральному классу

**\*\*Возвращает:\*\***

- Запрос с примененными фильтрами

"""

# Поиск по названию

if name:

    query = query.where(CelestialBody.name.ilike(name))

# Фильтр по типу

if body\_type:

    query = query.where(CelestialBody.type == body\_type)

# Фильтр по расстоянию

if min\_distance is not None:

    query = query.where(CelestialBody.distance\_from\_earth >= min\_distance)

if max\_distance is not None:

    query = query.where(CelestialBody.distance\_from\_earth <= max\_distance)

# Фильтр по звёздной величине

if min\_magnitude is not None:

    query = query.where(CelestialBody.apparent\_magnitude >= min\_magnitude)

if max\_magnitude is not None:

    query = query.where(CelestialBody.apparent\_magnitude <= max\_magnitude)

# Фильтр по спектральному классу

if spectral\_class:

    query = query.where(CelestialBody.spectral\_class == spectral\_class)

return query

## app/main.py

Содержание проекта

### main.py

```
"""
```

```
Главный файл приложения FastAPI.
```

```
Содержит конфигурацию и подключение маршрутов.
```

```
"""
```

```
from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse
from fastapi.exceptions import RequestValidationError
from starlette.exceptions import HTTPException as
StarletteHTTPException
from contextlib import asynccontextmanager
import logging
from datetime import datetime

from app.database import init_db
from app.routers import (
 celestial_bodies_router,
 astronomers_router,
 observations_router,
 auth_router
)

Настройка логирования
logging.basicConfig(
 level=logging.INFO,
 format="%(asctime)s - %(name)s - %(levelname)s - %
(message)s"
)
logger = logging.getLogger(__name__)

Lifespan context manager для управления жизненным циклом
приложения
```

```
@asynccontextmanager
async def lifespan(app: FastAPI):
 """
 Контекстный менеджер для управления жизненным циклом
 приложения.

 Выполняется при запуске и остановке приложения.
 """

 # Код при запуске приложения
 logger.info("🚀 Запуск приложения Astronomy API...")

 # Инициализация базы данных
 try:
 await init_db()
 logger.info("✅ База данных инициализирована")
 except Exception as e:
 logger.error(f"❌ Ошибка инициализации базы данных:
{e}")

 yield # Приложение работает

 # Код при остановке приложения
 logger.info("👋 Приложение остановлено")

Создание приложения FastAPI
app = FastAPI(
 title="Astronomy API",
 description="",
 API для управления базой данных астрономических объектов.

 ## Функциональность:
 - Управление небесными телами (планеты, звезды, галактики)
 - Управление информацией об астрономах
 - Запись и анализ наблюдений
 - Расширенный поиск и фильтрация
 - Аутентификация пользователей

 ## Технологии:
 - FastAPI 0.109.0
```

```
- SQLModel (SQLAlchemy 2.0 + Pydantic)
- PostgreSQL / SQLite
"""
version="1.0.0",
contact={
 "name": "Astronomy API Team",
 "email": "admin@astronomy-api.com"
},
license_info={
 "name": "MIT License",
 "url": "https://opensource.org/licenses/MIT"
},
lifespan=lifespan,
docs_url="/docs",
redoc_url="/redoc",
openapi_url="/openapi.json"
)

Настройка CORS (Cross-Origin Resource Sharing)
app.add_middleware(
 CORSMiddleware,
 allow_origins=["*"],
 allow_credentials=True,
 allow_methods=["*"],
 allow_headers=["*"],
)

Подключение маршрутов
app.include_router(celestial_bodies_router)
app.include_router(astronomers_router)
app.include_router(observations_router)
app.include_router(auth_router)

Корневой эндпоинт
@app.get(
 "/",
 summary="Корневой эндпоинт",
```

```
 description="Возвращает информацию о приложении"
)
 async def root():
 """Корневой эндпоинт приложения"""
 return {
 "message": "🔭 Astronomy API",
 "version": "1.0.0",
 "status": "running",
 "timestamp": datetime.utcnow().isoformat(),
 "docs": {
 "swagger": "/docs",
 "redoc": "/redoc",
 "openapi": "/openapi.json"
 }
 }
}

Эндпоинт здоровья приложения
@app.get(
 "/health",
 summary="Проверка здоровья",
 description="Проверяет работоспособность приложения и базы
 данных"
)
async def health_check():
 """Проверка здоровья приложения"""
 from app.database import engine

 try:
 async with engine.connect() as conn:
 await conn.execute("SELECT 1")
 database_status = "healthy"
 except Exception as e:
 database_status = f"unhealthy: {str(e)}"

 return {
 "status": "healthy" if database_status == "healthy" else
 "unhealthy",
 "database": database_status,
 "timestamp": datetime.utcnow().isoformat()
```

```
}

Обработчик ошибок валидации
@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc: RequestValidationError):
 """
 Обработчик ошибок валидации запросов.

 Возвращает понятные сообщения об ошибках валидации.
 """

 logger.error(f"Ошибка валидации: {exc.errors()}")
 return JSONResponse(
 status_code=422,
 content={
 "detail": "Ошибка валидации данных",
 "errors": exc.errors(),
 "body": exc.body
 }
)

Обработчик HTTP ошибок
@app.exception_handler(StarletteHTTPException)
async def http_exception_handler(request: Request, exc: StarletteHTTPException):
 """Обработчик HTTP ошибок"""
 logger.error(f"HTTP ошибка {exc.status_code}: {exc.detail}")
 return JSONResponse(
 status_code=exc.status_code,
 content={
 "detail": exc.detail,
 "status_code": exc.status_code
 }
)

Обработчик неожиданных ошибок
@app.exception_handler(Exception)
```

```
async def general_exception_handler(request: Request, exc: Exception):
 """Обработчик неожиданных ошибок"""
 logger.error(f"Неожиданная ошибка: {exc}", exc_info=True)
 return JSONResponse(
 status_code=500,
 content={
 "detail": "Внутренняя ошибка сервера",
 "message": str(exc)
 }
)

if __name__ == "__main__":
 import uvicorn

 uvicorn.run(
 "app.main:app",
 host="0.0.0.0",
 port=8000,
 reload=True,
 log_level="info"
)
```