МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА №44

КУРСОВАЯ РАБОТА (ПРОЕКТ)   
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподаватель |  |  |  | Д. В. Куртяник |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ |
| СОЗДАНИЕ API ДЛЯ АРЕНДЫ |
| по дисциплине: Основы программирования |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ гр. № | 4319 |  | 14.12.2024 |  | А. М. Емельянов |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#__RefHeading___Toc4123_790615146)

[1 Анализ требований к API аренды 5](#__RefHeading___Toc4132_790615146)

[1.1 Что такое API 5](#__RefHeading___Toc4134_790615146)

[1.2 Определение функциональных и нефункциональных требований 5](#__RefHeading___Toc4563_790615146)

[1.3 Постановка задач и определение ограничений 6](#__RefHeading___Toc4565_790615146)

[2 Проектирование структуры API 6](#__RefHeading___Toc4672_790615146)

[2.1 Выбор технологий и инструментов разработки 6](#__RefHeading___Toc4674_790615146)

[2.2 Проектирование структуры данных и базы данных 7](#__RefHeading___Toc4676_790615146)

[2.3 Разработка схемы маршрутов (эндпоинтов) 9](#__RefHeading___Toc4795_790615146)

[3. Реализация API для аренды 9](#__RefHeading___Toc4797_790615146)

[3.1 Настройка и развертывание окружения 9](#__RefHeading___Toc4799_790615146)

[Зачем нужен Docker Compose: 11](#__RefHeading___Toc4940_790615146)

[3.2 Реализация основных модулей API 11](#__RefHeading___Toc4942_790615146)

[Далее разберёмся с аргументами функции. 14](#__RefHeading___Toc4949_790615146)

[Далее разберёмся с логикой обработки. 15](#__RefHeading___Toc4951_790615146)

[Что возращает метод. 15](#__RefHeading___Toc4953_790615146)

[3.3 Обеспечение валидации данных и обработки ошибок 18](#__RefHeading___Toc4955_790615146)

[3.3.1. Что такое Pydantic схемы? 19](#__RefHeading___Toc4957_790615146)

[3.3.2. Зачем использовать Pydantic схемы? 19](#__RefHeading___Toc4959_790615146)

[3.3.3. Как работает Pydantic схема в примере? 20](#__RefHeading___Toc4961_790615146)

[3.3.4. Преимущества применения Pydantic схем: 21](#__RefHeading___Toc4963_790615146)

[3.3.5. Пример работы с запросом: 22](#__RefHeading___Toc4965_790615146)

[ЗАКЛЮЧЕНИЕ 23](#__RefHeading___Toc4967_790615146)

[ПРИЛОЖЕНИЕ A Код для Docker compose файл 24](#__RefHeading___Toc4944_790615146)

[Приложение Б код класса BaseDAO 25](#__RefHeading___Toc4969_790615146)

# ВВЕДЕНИЕ

Современные технологии играют ключевую роль в автоматизации процессов во многих сферах человеческой деятельности. Одной из таких сфер является аренда — будь то строительное оборудование, транспортные средства или жилые помещения. Успешное управление такими процессами требует создания эффективных и надёжных решений, обеспечивающих быструю обработку данных и удобный доступ для пользователей.

API (Application Programming Interface) — это программный интерфейс, который позволяет взаимодействовать между различными системами. Создание API для аренды предоставляет возможность централизовать обработку запросов, интегрировать внешние системы и улучшить пользовательский опыт за счёт высокой скорости и доступности данных. Такие интерфейсы позволяют автоматизировать процесс аренды, снижая вероятность ошибок и повышая оперативность работы.

Целью данной курсовой работы является разработка программного интерфейса (API), который обеспечит выполнение ключевых задач в системе аренды. В рамках работы будут рассмотрены современные подходы к проектированию API, а также продемонстрирован процесс разработки, тестирования и развертывания решения.

Создание API для аренды требует понимания архитектуры веб-приложений, работы с базами данных, а также использования современных инструментов и фреймворков. В качестве основы будет использован язык программирования Python с применением фреймворка FastAPI, который зарекомендовал себя как надёжное и производительное решение для создания RESTful API.

Практическое значение данной работы заключается в том, что разработанное API может быть адаптировано для различных систем аренды, а его функционал легко расширяется под требования конкретных задач. Итоговый результат будет представлен в виде полностью функционирующего интерфейса с задокументированными эндпоинтами, поддерживающего основные операции CRUD (Create, Read, Update, Delete).

# 1 Анализ требований к API аренды

## 1.1 Что такое API

Для начала напомню, что API (Application Programming Interface) — это интерфейс программирования приложений, который предоставляет набор определённых правил и механик взаимодействия между различными программными компонентами. С помощью API разработчики могут строить приложения, интегрировать их с внешними системами, обеспечивая передачу данных и выполнение различных операций. API обычно представлены в виде REST, GraphQL, SOAP или других подходов, среди которых REST является наиболее популярным для веб-приложений. В своей программе мы также пользуемся принципами REST.

API для системы бронирования — это интерфейс, который предоставляет функциональность для управления бронированиями (создание, обновление, удаление), а также позволяет интегрировать эту систему с внешними клиентами (мобильными приложениями, сторонними сервисами).

## 1.2 Определение функциональных и нефункциональных требований

Функциональные требования описывают, какие задачи должна выполнять система. Это определение поведения системы при взаимодействии с пользователем или другим компонентом. Например, создание, удаление или изменение аренды — это примеры функциональных требований. Они позволяют понять, какую пользу приносит система пользователям и какие основные функции она должна предоставлять.

Нефункциональные требования описывают, каким образом система должна выполнять свои функции. Они включают в себя такие аспекты, как производительность, надёжность, безопасность, масштабируемость и удобство использования. Например, требование, чтобы система обрабатывала запросы с задержкой не более 200 мс, относится к нефункциональным требованиям.[1]

Функциональные требования:

* Возможность создания новой аренды;
* Получение информации о существующих арендах пользователя;
* Изменение деталей аренды;
* Удаление аренды из базы данных;
* Авторизация пользователей.

Нефункциональные требования:

* Быстродействие при обработке запросов;
* Надёжность и отказоустойчивость системы;
* Удобство интеграции с другими системами.

## 1.3 Постановка задач и определение ограничений

Основные задачи:

* + - Разработать структуру базы данных для хранения данных о аренды;
    - Реализовать маршруты API для выполнения операций с арендой;
    - Обеспечить безопасность данных через авторизацию и валидацию запросов.

Ограничения:

* + Используемый стек технологий — Python, FastAPI, PostgreSQL, Redis, Celery, SQLAlchemy, Docker.
  + Проект ориентирован на локальную среду разработки.

# 2 Проектирование структуры API

## 2.1 Выбор технологий и инструментов разработки

Для реализации API выбраны следующие инструменты:

* + FastAPI — асинхронный фреймворк для создания API на Python.
  + PostgreSQL — реляционная база данных для хранения данных.
  + SQLAlchemy — инструмент для работы с базой данных.
  + Redis – NoSQL база данных для кеширования данных и правильной работы Celery.
  + Celery – асинхронная очередь задач.
  + Docker — для контейнеризации приложения.

## 2.2 Проектирование структуры данных и базы данных

Базой данных (БД) называется организованная в соответствии с определенными правилами и поддерживаемая в памяти компьютера совокупность сведений об объектах, процессах, событиях или явлениях, относящихся к некоторой предметной области, теме или задаче [2]. Она организована таким образом, чтобы обеспечить информационные потребности пользователей, а также удобное хранение этой совокупности данных, как в целом, так и любой ее части.

Реляционные базы данных состоят из таблиц. Каждая таблица состоит из столбцов (их называют полями или атрибутами) и строк (их называют записями или кортежами). Таблицы в реляционных базах данных обладают рядом свойств.

1. В таблице не может быть двух одинаковых строк. В математике таблицы, обладающие таким свойством, называют отношениями-по-английски relation, отсюда и название – реляционные.
2. Столбцы располагаются в определенном порядке, который создается при создании таблицы. В таблице может не быть ни одной строки, но обязательно должен быть хотя бы один столбец.
3. У каждого столбца есть уникальное имя (в пределах таблицы), и все значения в одном столбце имеют один тип (число, текст, дата...).
4. На пересечении каждого столбца и строки может находиться только атомарное значение (одно значение, не состоящее из группы значений). Таблицы, удовлетворяющие этому условию, называют нормализованными.

Любая система управления базами данных (СУБД) позволяет выполнять следующие операции с данными:

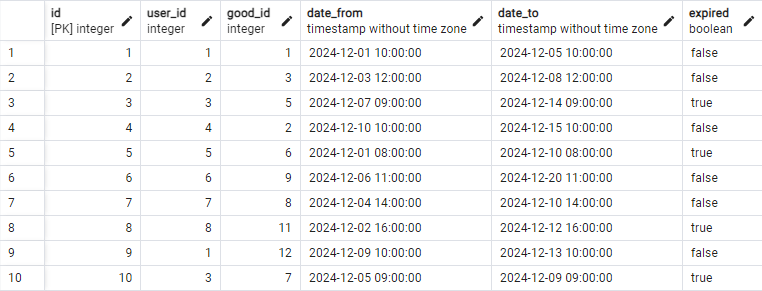
* добавление записей в таблицы;
* удаление записей из таблицы;
* обновление значений некоторых полей в одной или нескольких записях в таблицах БД;
* поиск одной или нескольких записей, удовлетворяющих заданному условию.

Основные таблицы нашей базы данных:

* + Categories
  + Goods
  + Users
  + Rentals

Разберёмся для чего нужна каждая из них. 1) Categories – эта таблица содержит данные о категории товара (например Машины, Строительные инструменты) и описание этой категории. 2) Goods содержит id категории товара из таблицы Categories, название товара и его оставшееся количество. 3) Users заполнена данными о пользователях: их почты и хешированные пароли. 4) Rentals – последняя таблица базы данных. В ней хранятся данные о пользователях, а именно их id, чтобы пользователь мог легко получить данные о том, какие товары у него сейчас арендованы. Также в таблице хранятся данные о том с какой даты и по какую пользователь арендует товар.

Пример того как выглядят данные в PostgreSQL (рисунок 1).

Рисунок 1 — Данные в PostgreSQL таблице rentals

## 2.3 Разработка схемы маршрутов (эндпоинтов)

Пример маршрутов:

* + POST /goods/add\_good — добавление нового товара для определённой категории;
  + GET /rent/my\_rents — получение информации о своих арендах;
  + DELETE /goods/good/delete/{good\_id}` — удаление товара.

# 3. Реализация API для аренды

## 3.1 Настройка и развертывание окружения

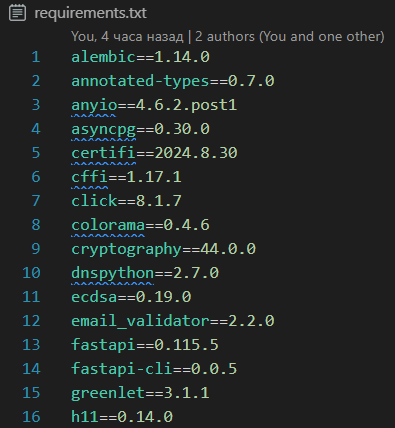
1. Установите необходимые зависимости:

pip install fastapi[all] uvicorn sqlalchemy python-jose

Для создания первого эндпоинта данного набора библиотек хватит. Далее по мере создания проекта количество использованных библиотек будет увеличиваться, поэтому в репозитории будем пересоздавать файл с необходимыми для работы библиотеки с помощью команды:

pip freeze > requirements.txt

В результате получается файл, в котором хранятся названия библиотек. То, как выглядят данные внутри файла можно увидеть на рисунке 2.

Рисунок 2 — файл requirements.txt

Вместо requirements можно вставить своё название файла в котором будет список библиотек, которые используются в проекте.

2. Настройте Docker Compose для работы с PostgreSQL:

**Docker Compose** — это инструмент для определения и управления многоконтейнерными Docker-приложениями. Он позволяет описать конфигурацию всех контейнеров, сетей и томов в одном YAML-файле (docker-compose.yml), чтобы проще запускать и управлять приложениями, состоящими из нескольких сервисов.

Контейнеры — это изолированные окружения, в которых можно запускать приложения и их зависимости (например можно запустить образ Python). Они позволяют упаковать приложение и все его нужные компоненты (например, библиотеки, настройки, файлы) в единую единицу, которая будет одинаково работать на любой машине, где установлен Docker. Контейнеры используют ядро операционной системы хост-машины, что делает их более легкими по сравнению с виртуальными машинами.

### Зачем нужен Docker Compose:

1. **Упрощение конфигурации**: Вместо того чтобы запускать несколько контейнеров вручную с разными командами, можно описать все необходимые контейнеры и их параметры в одном файле. Это делает процесс настройки приложения и его зависимостей более удобным.
2. **Легкость в запуске**: С помощью одной команды docker-compose up можно развернуть весь проект с его зависимостями. Например, если ваш проект зависит от базы данных (например, PostgreSQL), вы можете настроить и запустить как приложение, так и базу данных одновременно.
3. **Масштабирование и тестирование**: Docker Compose позволяет легко масштабировать сервисы (например, создать несколько экземпляров базы данных или веб-серверов) и использовать однообразные настройки для разработки и тестирования.
4. **Изоляция среды**: С помощью Docker Compose можно создать изолированную среду для каждого проекта, минимизируя конфликты между различными зависимостями.

Код docker-compose файла представлен в приложении А

## 3.2 Реализация основных модулей API

1) Эндпоинт для создания аренды

@router.post("/add\_rent")

async def add\_rent(good\_id: int,

date\_from: date,

date\_to: date,

expired: bool,

user: Users = Depends(get\_current\_user)):

if not user:

raise TokenAbsentException

if date\_from > date\_to:

raise IncorrectDateException

good\_amount = await GoodsDAO.find\_by\_id(good\_id)

if good\_amount and good\_amount.amount > 0:

new\_rent = await RentalsDAO.add(

user\_id=user.id,

good\_id=good\_id,

date\_from=date\_from,

date\_to=date\_to,

expired=expired

)

await GoodsDAO.update(id=good\_id, field="amount", data=good\_amount.amount - 1)

else:

raise LackOfGoodException

Рассмотрим как он работает. Этот код представляет собой обработчик POST-запроса в веб-приложении. Основной задачей этого обработчика является добавление нового арендного соглашения для товара в базе данных, с учётом нескольких проверок. Над самой функцией у нас указан декоратор. Внутри этого декоратора мы указываем path, то есть наш эндпоинт или же маршрут. На вход принимаются все значения, которые нужны для добавления аренды в базу данных:

* + good\_id: int – идентификатор товара, который пользователь хочет арендовать. Это число передаётся в теле запроса, как и остальные переменные
  + date\_from: date – дата с которой начинается аренда
  + date\_to: date – дата до которой продлится аренда
  + expired: bool – флаг, который показывает закончилась ли аренда или нет.
  + user: Users = Depends(get\_current\_user) — Depends — это встроенная функция в FastAPI, которая помогает получить значение из другой функции и передать её значение в переменную. В функции get\_current\_user происходят различные проверки cookie, которую мы выдаём пользователю при авторизации. Если эта cookie правильная и данные в ней содержаться правильные, то мы даём пользователю взаимодействовать с эндпоинтом

Когда мы заходим в функцию, то происходит ряд проверок перед тем как мы добавим данные в PostgreSQL. Сначала проверяется, что пользователь действительно вошел в свой аккаунт (если произойдут ошибки во время проверки, то get\_current\_user выдаст ошибку или ничего не отдаст). Следующий шаг — проверка, что дата начала аренды (date\_from) не больше даты окончания аренды (date\_to). Если это условие нарушено, выбрасывается исключение IncorrectDateException. Далее выполняется запрос к базе данных с помощью GoodsDAO.find\_by\_id(good\_id), который находит товар по идентификатору good\_id. Возвращается объект товара, в котором содержится информация о его количестве (amount). Если товар найден и его количество больше 0 (то есть товар доступен для аренды), выполняется дальнейшее добавление аренды. Если товар отсутствует или его количество равно 0, выбрасывается исключение LackOfGoodException, что означает отсутствие товара для аренды.

Если товар доступен, создается новое арендное соглашение с помощью функции RentalsDAO.add. Эта функция добавляет запись о новой аренде в базу данных, где передаются следующие параметры:

* + user\_id=user.id: Идентификатор пользователя, который арендует.
  + good\_id=good\_id: Идентификатор товара, который арендуется.
  + date\_from=date\_from: Дата начала аренды.
  + date\_to=date\_to: Дата окончания аренды.
  + expired=expired: Флаг, указывающий, является ли аренда истекшей.

После того как аренда успешно добавлена, количество товара в базе данных уменьшается на 1 (так как товар был арендован). Для этого используется функция GoodsDAO.update, которая обновляет запись о товаре, уменьшая его количество на единицу: good\_amount.amount - 1.

2) Эндпоинт для получения информации о бронировании

from fastapi import APIRouter, Depends

from datetime import date

from app.users.models import Users

from app.exceptions import TokenAbsentException, IncorrectDateException, LackOfGoodException

from app.users.dependencies import get\_current\_user

from app.rentals.dao import RentalsDAO

from app.goods.dao import GoodsDAO

router = APIRouter(

prefix="/rent",

tags=["Аренда"]

)

@router.get("/my\_rents")

async def get\_my\_rents(user: Users = Depends(get\_current\_user)):

rents = await RentalsDAO.find\_all(user\_id=user.id)

return rents

Этот метод представляет собой обработчик GET-запроса, который возвращает список всех арендуемых товаров текущего пользователя. Рассмотрим как он работает.

Сначала разберёмся с тем как работает роутер.

* + Декоратор @router.get("/my\_rents") указывает, что функция будет обрабатывать GET-запрос по маршруту /rent/my\_rents. Это значит, что когда пользователь выполнит GET-запрос по этому пути, будет вызвана эта функция.
  + tags=["Аренда"] — это метка для документации, которая группирует эндпоинты, связанные с арендами, в одну категорию для удобства.

### **Далее разберёмся с аргументом функции.**

* user: Users = Depends(get\_current\_user) — как и было описано в предыдущем методе. Это функция, которая помогает определить действительно ли пользователь вошёл в свою учётную запись.

### **Далее разберёмся с логикой обработки.**

* rents = await RentalsDAO.find\_all(user\_id=user.id):
  + Этот запрос к базе данных с помощью DAO (Data Access Object) RentalsDAO.find\_all ищет все аренды, связанные с данным пользователем.
  + Метод find\_all получает все записи аренды, где user\_id соответствует ID текущего пользователя (извлекается через user.id).
  + Параметр await указывает, что метод работает асинхронно, и выполнение будет приостановлено до получения результата. Это важно, так как запросы к базе данных могут занять время.

### **И наконец, что возращает метод.**

* После выполнения запроса RentalsDAO.find\_all переменная rents будет содержать список всех арендуемых товаров текущего пользователя.
* В FastAPI по умолчанию возвращаемое значение из функции будет автоматически преобразовано в формат JSON и отправлено клиенту как ответ на запрос.

3) Стоит также рассмотреть базовый класс, который используется как макет для классов каждой из моделей для взаимодействия с PostgreSQL:

Наш базовый класс - это **DAO (Data Access Object)** — это шаблон проектирования, используемый в разработке программного обеспечения для управления доступом к данным. Он служит посредником между приложением и системой хранения данных (например, базой данных). Основная идея DAO заключается в абстрагировании операций работы с данными, таких как чтение, запись, обновление и удаление, чтобы приложение не зависело от конкретной технологии или структуры базы данных. В нашем случае мы создали класс BaseDAO. Далее мы разберёмся как работает его код.

### Основные особенности DAO:

1. **Инкапсуляция доступа к данным:** DAO — это изолированный класс который с помощью описанной логики взаимодействует с базой данных от бизнес-логики приложения. Это упрощает изменения в структуре хранения данных без необходимости модифицировать остальной код.
2. **Повышение читаемости и поддержки кода:** Благодаря использованию DAO происходит чёткое разделение слоёв приложения, что в свою очередь делает архитектуру более структурированной. Это упрощает внесение изменений и добавление нового функционала в API.
3. **Снижение дублирования кода:** Общие операции с данными выносятся в DAO, что устраняет повторяющиеся участки кода, что соответствует принципу DRY (Don’t Repeat Yourself).

Полный код метода можно посмотреть в примечании Б.

* + - * + **find\_by\_id - Этот метод находит один объект модели по его идентификатору. В функцию передаём один аргумент — model\_id (ID записи, которую мы хотим). Порядок дейсствий: открывается сессия с базой данных с помощью** async\_session\_maker**. Далее строится SQL-запрос с использованием SQLAlchemy, чтобы найти запись по полю** id **в таблице модели (**cls.model.\_\_table\_\_.columns**). Запрос выполняется с помощью** await session.execute(query)**. Метод возвращает либо найденный объект, либо** None**, если запись с таким ID не существует, с помощью** result.mappings().one\_or\_none()**.**

**Код:**

**@classmethod**

**async def find\_by\_id(cls, model\_id: int):**

**async with async\_session\_maker() as session:**

**query = select(cls.model.\_\_table\_\_.columns).filter\_by(id=model\_id)**

**result = await session.execute(query)**

**return result.mappings().one\_or\_none()**

* + **find\_one\_or\_none - Этот метод находит одну запись по любым переданным аргументам, например, по имени или другому уникальному полю. Аргументы метода -** \*\*kwargs, то есть **произвольные параметры для фильтрации, например,** find\_one\_or\_none(name='Item1')**. Порядок действий: Открывается сессия с базой данных. Создаётся запрос с фильтром по переданным полям через** filter\_by(\*\*kwargs)**. Запрос выполняется, и возвращается либо один объект, либо** None**, если запись не найдена.**

**Код:**

**@classmethod**

**async def find\_one\_or\_none(cls, \*\*kwargs):**

**async with async\_session\_maker() as session:**

**query = select(cls.model.\_\_table\_\_.columns).filter\_by(\*\*kwargs)**

**result = await session.execute(query)**

**return result.mappings().one\_or\_none()**

* + **find\_all** - Этот метод находит все записи, которые соответствуют определённым критериям. **Аргументы -** \*\*filter\_by — произвольные параметры для фильтрации записей. Порядок действий: Открывается сессия с базой данных. Строится запрос с фильтрацией, используя переданные аргументы, аналогично методу find\_one\_or\_none. Запрос выполняется, и возвращается список всех найденных объектов с помощью result.scalars().all().

**Код:**

**@classmethod**

**async def find\_all(cls, \*\*filter\_by):**

**async with async\_session\_maker() as session:**

**query = select(cls.model).filter\_by(\*\*filter\_by)**

**result = await session.execute(query)**

**return result.scalars().all()**

* + add - Этот метод добавляет новую запись в таблицу модели. **Аргументы -** \*\*data — данные для вставки в таблицу. Например, это может быть словарь с данными, как name='Item1', description='A test item'. Порядок действий: открывается сессия с базой данных. Строится запрос на вставку данных с помощью insert(cls.model).values(\*\*data), который вставляет переданные данные в таблицу модели. Запрос выполняется с помощью await session.execute(query). После выполнения запроса вызывается await session.commit() для коммита транзакции. Метод возвращает вставленную запись.

**Код:**

**@classmethod**

**async def add(cls, \*\*data):**

**async with async\_session\_maker() as session:**

**query = insert(cls.model).values(\*\*data).returning(cls.model)**

**result = await session.execute(query)**

**await session.commit()**

**return result.scalar()**

* + Update - Этот метод обновляет значение конкретного поля у записи по её ID. **Аргументы -** id — ID записи, которую нужно обновить; field — название поля, которое нужно обновить; data — новое значение этого поля. Порядок действий: Открывается сессия с базой данных. Строится запрос на обновление, который обновляет поле field у записи с указанным id. Если запрос успешно выполнен, то вызывается await session.commit() для фиксации изменений. Метод возвращает обновлённую запись, либо None, если запрос не дал результатов.

**Код:**

**@classmethod**

**async def update(cls, id: int, field: str, data):**

**async with async\_session\_maker() as session:**

**query = update(cls.model).where(cls.model.id == id).values({field: data}).returning(cls.model)**

**if query is not None:**

**updated = await session.execute(query)**

**await session.commit()**

**return updated**

**else:**

**return None**

* + Delete - Этот метод удаляет запись по её ID. **Аргументы -** id — ID записи, которую нужно удалить. Порядок действий: Открывается сессия с базой данных. Строится запрос на удаление записи, которая имеет указанный id. Запрос выполняется с помощью await session.execute(query). После этого вызывается await session.commit() для подтверждения удаления записи.

**Код:**

**@classmethod**

**async def delete(cls, id: int):**

**async with async\_session\_maker() as session:**

**query = delete(cls.model).where(cls.model.id == id)**

**if query is not None:**

**await session.execute(query)**

**await session.commit()**

## 3.3 Обеспечение валидации данных и обработки ошибок

В нашем проекте используются Pydantic схемы, которые представляют собой описание данных, которые ожидаются в теле POST-запроса. Использование Pydantic схем в FastAPI позволяет эффективно валидировать и сериализовать данные, а также автоматизировать процесс их обработки. Давайте подробнее разберём, что такое Pydantic схемы, зачем они нужны и как они работают и рассмотрим одну из Pydantic схем.

### **3.3.1. **Что такое Pydantic схемы?****

Pydantic — это библиотека для валидации данных и их сериализации, которая использует Python type hints (аннотации типов). Она позволяет описывать структуры данных с типами (как правило — это классы), а затем автоматически проверять, соответствуют ли переданные данные этим типам, и при необходимости преобразовывать их в нужный формат.

Если говорить о Pydantic схеме, то мы имеем в виду класс, наследующийся от BaseModel. Этот класс описывает структуру данных, определяя поля, их типы и возможные ограничения(например, длину строки). В FastAPI Pydantic схемы используются для:

* Валидации данных, которые приходят в запросе.
* Преобразования данных в нужный формат (например, в объекты Python).

### **3.3.2. **Зачем использовать Pydantic схемы?****

Использование Pydantic схем в FastAPI помогает решить несколько задач:

* **Валидация данных**: FastAPI автоматически проверяет, соответствуют ли данные, переданные в теле запроса, определённой схеме. Если данные не проходят валидацию (например, типы данных не совпадают или обязательные поля отсутствуют), FastAPI вернёт ошибку с подробным описанием.
* **Автоматическое преобразование типов**: Pydantic позволяет конвертировать строки в нужные типы, например, преобразовать строковое представление числа в целое число. Это намного упрощает работу с запросами.
* **Поддержка документации API**: Pydantic схемы интегрируются с документацией FastAPI, автоматически создавая описание ожидаемых входных данных в интерфейсе Swagger UI.

### **3.3.3. **Как работает Pydantic схема в примере?****

В данном примере используется Pydantic схема AddGood для описания данных товара, которые должны быть переданы в теле POST-запроса для создания нового товара.

#### Описание схемы AddGood:

class AddGood(BaseModel):

category\_id: int

name: str

description: str

price\_per\_day: int

amount: int

Схема определяет 5 полей:

* category\_id: целое число, идентификатор категории товара.
* name: строка, название товара.
* description: строка, описание товара.
* price\_per\_day: целое число, цена за день аренды товара.
* amount: целое число, количество товара.

Каждое из этих полей указано с типом данных. Как уже было описано, это позволяет FastAPI автоматически проверять, что данные, переданные в запросе, соответствуют этим типам:

* Например, если в запросе для поля category\_id будет передано значение типа str вместо int, FastAPI вернёт ошибку с подробным описанием того, что ожидается целое число.
* Если в запросе будет отсутствовать одно из обязательных полей, FastAPI также вернёт ошибку.

#### Применение схемы в маршруте:

@router.post("/add\_good")

async def add\_good(good\_info: AddGood):

try:

new\_good = await GoodsDAO.add(\*\*dict(good\_info))

except Exception:

return "Ошибка"

return new\_good

1. **Параметр** good\_info: AddGood:
   * FastAPI автоматически извлекает данные из тела POST-запроса и пытается преобразовать их в объект типа AddGood. Для этого он использует Pydantic, который проверяет, что данные соответствуют типам, определённым в схеме.
   * Если данные соответствуют схеме, FastAPI передаст объект good\_info в метод add\_good, который будет уже содержать валидированные данные в виде Python объекта.
2. await GoodsDAO.add(\*\*dict(good\_info)):
   * После того как данные валидированы и переданы в функцию, они конвертируются в словарь с помощью dict(good\_info). Это необходимо, так как метод GoodsDAO.add() ожидает словарь данных для добавления в базу данных.
   * \*\*dict(good\_info) передаёт данные в виде ключей и значений в метод add, который добавляет товар в базу данных.
3. **Обработка исключений**:
   * В случае возникновения ошибки при добавлении товара (например, ошибка базы данных), будет поймано исключение и возвращено сообщение "Ошибка". В реальном приложении стоит использовать более конкретные ошибки и логировать их для диагностики.

### **3.3.4. **Преимущества применения Pydantic схем:****

* **Удобство и безопасность**: С помощью Pydantic вы автоматически получаете проверку типов и данных, что снижает вероятность ошибок и повышает безопасность приложения.
* **Чистота кода**: Применение Pydantic схем позволяет изолировать логику работы с данными от бизнес-логики, упрощая код и делая его более читаемым.
* **Документация**: Pydantic автоматически генерирует описание ожидаемых данных в Swagger UI, что улучшает понимание API для разработчиков и пользователей.

### **3.3.5. **Пример работы с запросом**:**

Если вы отправите POST-запрос на /add\_good с телом json:

{

"category\_id": 1,

"name": "Test Good",

"description": "A good for testing",

"price\_per\_day": 100,

"amount": 10

}

FastAPI автоматически проверит, что все поля соответствуют типам, описанным в AddGood. Если данные корректны, они будут переданы в метод add\_good, а товар будет добавлен в базу данных.

Если, например, вы передадите:

{

"category\_id": "1", // некорректный тип

"name": "Test Good",

"description": "A good for testing",

"price\_per\_day": 100,

"amount": 10

}

FastAPI вернёт ошибку с сообщением, что category\_id должен быть целым числом, и запрос не будет обработан.

## 3.4 Использование Celery и Redis для обработки фоновых задач в системе аренды

Для реализации функционала, связанного с обработкой уведомлений и других долгосрочных операций в системе аренды, используется связка Celery и Redis. Такой подход обеспечивает выполнение задач в фоновом режиме, что позволяет не блокировать основной поток обработки запросов и повышает общую производительность системы.

Celery играет ключевую роль в обработке фоновых задач. В рамках системы аренды Celery используется для:

* Отправки уведомлений пользователям о просрочке оплаты бронирований;
* Обработки периодических задач, таких как автоматическое завершение бронирований с истёкшим сроком;

Благодаря асинхронной архитектуре Celery задачи добавляются в очередь и обрабатываются отдельными воркерами, что позволяет системе эффективно распределять нагрузку.

В проекте Redis используется как брокер сообщений для Celery. Это позволяет надёжно хранить и передавать задачи от основного приложения к воркерам. Выбор Redis обусловлен его высокой скоростью, простотой настройки и надёжностью. Он обеспечивает:

* Мгновенное добавление и получение задач из очереди;
* Гибкость масштабирования за счёт возможности подключения дополнительных воркеров;
* Возможность работы с большими объёмами данных благодаря хранению информации в оперативной памяти.

Использование Celery и Redis совместно позволяет:

1. **Повысить производительность системы:** долгосрочные операции не блокируют основной поток приложения, что ускоряет обработку пользовательских запросов.
2. **Обеспечить масштабируемость:** при увеличении числа пользователей можно легко добавить больше воркеров для обработки задач.
3. **Снизить вероятность ошибок:** фоновые задачи выполняются изолированно, что уменьшает риск сбоев в основном приложении.

Celery официально не поддерживает Windows как платформу для запуска воркеров из-за ограничений в многопоточности и взаимодействии процессов. Для преодоления этой проблемы в проекте используется Docker, который предоставляет среду для запуска Celery на базе Linux.

# ЗАКЛЮЧЕНИЕ

В рамках курсовой работы был создан API для аренды, обеспечивающий основные операции с данными. Работа включает проектирование, реализацию и ручное тестирование функционала. Данное API предоставляет возможность легко адаптировать его под различные сценарии аренды, что делает его универсальным инструментом для автоматизации подобных процессов.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. [Функциональные и нефункциональные требования](https://sky.pro/wiki/sql/funkcionalnye-i-nefunkcionalnye-trebovaniya/): <https://sky.pro/wiki/sql/funkcionalnye-i-nefunkcionalnye-trebovaniya/>
2. Пирогов, В. Ю. Информационные системы и базы данных: организация и проектирование / В. Ю. Пирогов. – СПб.: БХВ-Петербург, 2009. – 528 с.

# ПРИЛОЖЕНИЕ A Код для Docker compose файл

services:

db:

image: postgres:15

container\_name: rent\_db

env\_file:

- .env

environment:

- POSTGRES\_PASSWORD=postgres

- POSTGRES\_DB=rent

volumes:

- postgresdata:/var/lib/postgresql/data

redis:

image: redis:7

container\_name: booking\_redis

app:

build:

context: .

container\_name: rent\_app

env\_file:

- .env

ports:

- "8448:8000"

command: sh -c "alembic upgrade head && gunicorn app.main:app --workers 4 --worker-class uvicorn.workers.UvicornWorker --bind=0.0.0.0:8000"

depends\_on:

- db

celery:

build:

context: .

container\_name: booking\_celery

command: sh -c "celery --app=app.tasks.celery\_app:celery worker -l INFO"

env\_file:

- .env-non-dev

depends\_on:

- redis

volumes:

postgresdata:

# Приложение Б код класса BaseDAO

from sqlalchemy import insert, delete, select, update

from app.database import async\_session\_maker

class BaseDAO:

model = None

@classmethod

async def find\_by\_id(cls, model\_id: int):

async with async\_session\_maker() as session:

query = select(cls.model.\_\_table\_\_.columns).filter\_by(id=model\_id)

result = await session.execute(query)

return result.mappings().one\_or\_none()

@classmethod

async def find\_one\_or\_none(cls, \*\*kwargs):

async with async\_session\_maker() as session:

query = select(cls.model.\_\_table\_\_.columns).filter\_by(\*\*kwargs)

result = await session.execute(query)

return result.mappings().one\_or\_none()

@classmethod

async def find\_all(cls, \*\*filter\_by):

async with async\_session\_maker() as session:

query = select(cls.model).filter\_by(\*\*filter\_by)

result = await session.execute(query)

return result.scalars().all()

@classmethod

async def add(cls, \*\*data):

async with async\_session\_maker() as session:

query = insert(cls.model).values(\*\*data).returning(cls.model)

result = await session.execute(query)

await session.commit()

return result.scalar()

@classmethod

async def update(cls, id: int, field: str, data):

async with async\_session\_maker() as session:

query = update(cls.model).where(cls.model.id == id).values({field: data}).returning(cls.model)

if query is not None:

updated = await session.execute(query)

await session.commit()

return updated

else:

return None

@classmethod

async def delete(cls, id: int):

async with async\_session\_maker() as session:

query = delete(cls.model).where(cls.model.id == id)

if query is not None:

await session.execute(query)

await session.commit()