МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«Санкт-Петербургский государственный университет аэрокосмического приборостроения»

кАФЕДРА 33

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподаватель |  |  |  | К.А.Жиданов |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

ОТЧЕТ  
О ЛАБОРАТОРНОЙ РАБОТЕ

|  |
| --- |
| ИТОГОВАЯ работа№1 |
| по дисциплине: Технологии и методы программирования |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент гр. № | 3331 |  |  |  | М.Ю.Брыкин |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2025

## **Цель работы**

Создание Web-Сервиса для регистрации и создания задач, интегрированные с Telegram ботом. С использованием ИИ.

**Стек технологий**

* **Aiogram** – Асинхронная библиотека для создания Telegram-ботов на Python, основанная на asyncio, для удобной обработки сообщений и взаимодействия с Telegram API.
* **VENV** – Виртуальное окружение для пакетов.
* **.env** – Виртуальное окружение для секретов.
* **Aiosqlite** – Асинхронная библиотека для работы с SQLite, предоставляющая удобный интерфейс для выполнения асинхронных запросов к базе данных.
* **Alembic** – Инструмент для управления миграциями базы данных, интегрированный с SQLAlchemy, для автоматизации изменений схемы базы.
* **FastAPI** – Высокопроизводительный фреймворк для создания асинхронных веб-сервисов и API на Python, с автоматической генерацией документации (OpenAPI/Swagger).
* **fastapi-users** – Плагин для FastAPI, упрощающий реализацию аутентификации и управления пользователями (регистрация, авторизация, роли).
* **Jinja2** – Шаблонизатор для генерации динамических HTML-страниц в FastAPI, позволяющий создавать серверный рендеринг без использования JavaScript.
* **Pydantic** – Библиотека для валидации данных и создания схем API, а также для управления конфигурацией приложения через типизированные модели.
* **SQLAlchemy** – ORM-библиотека для работы с реляционными базами данных, обеспечивающая удобный доступ к данным через объектно-ориентированный интерфейс.
* **PyJWT** – Библиотека для создания и верификации JWT-токенов, используемых для безопасной аутентификации и передачи данных между клиентом и сервером.
* **Uvicorn** – ASGI-сервер для запуска асинхронных веб-приложений, преобразующий HTTP-запросы в вызовы Python, совместимый с FastAPI.
* **Bootstrap** – CSS-фреймворк для создания адаптивных и стилизованных пользовательских интерфейсов, упрощающий разработку фронтенда.

**Ход работы**

1) Создание бота через botfather

2) Создание Web сервиса на Fastapi

3) Создание моделей для БД

4) Миграция моделей

5) Создание Api для auth

6) Создание Api для чтения и создавание задач

7) Написание шаблона на Jinja2

## **Ключевые файлы**

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.1 – Структура приложения (1)

Изображение выглядит как текст, снимок экрана, дизайн

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.2 – Структура приложения (2)

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.3 – Api для auth

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.4 – Api для tasks

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.5 – confing

Изображение выглядит как текст, снимок экрана, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.6 – Пребаза для создания id в каждой записи БД и асинхронный генератор сессии БД

Изображение выглядит как текст, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.7 –Механизм регистрации и авторизации (1)

Изображение выглядит как текст, снимок экрана, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.8 –Механизм регистрации и авторизации (2)

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.9 – Модель задач

Изображение выглядит как снимок экрана, текст, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.10 – Модель user с обратной связью по задачам

Изображение выглядит как текст, снимок экрана, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.11 – Модель SQL ALCHEMY User

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.12 – Схема для Api чтения и записи задач

|  |
| --- |
| from fastapi import APIRouter, Request, Form, Depends, Response from fastapi.responses import HTMLResponse, RedirectResponse from fastapi.templating import Jinja2Templates import aiohttp import logging  logging.basicConfig(level=logging.INFO) logger = logging.getLogger(\_\_name\_\_)  router = APIRouter(tags=["auth\_views"])  templates = Jinja2Templates(directory="app/template")  @router.get("/", response\_class=HTMLResponse) async def register\_page(request: Request):  return templates.TemplateResponse(  "register.html",  {"request": request, "error": None, "success": None}  )  @router.post("/", response\_class=HTMLResponse) async def register\_user(  request: Request,  email: str = Form(...),  password: str = Form(...), ):  async with aiohttp.ClientSession() as session:  try:  async with session.post(  'http://localhost:8000/auth/register',  json={  'email': email,  'password': password,  'is\_active': True,  'is\_superuser': False,  'is\_verified': False  }  ) as reg\_response:  logger.info(f"Register response: {reg\_response.status} {await reg\_response.text()}, headers: {reg\_response.headers}")  if reg\_response.status == 201:  async with session.post(  'http://localhost:8000/auth/jwt/login',  data={  'username': email,  'password': password  },  headers={'Content-Type': 'application/x-www-form-urlencoded'}  ) as login\_response:  logger.info(f"Auto-login response: {login\_response.status} {await login\_response.text()}, headers: {login\_response.headers}")  if login\_response.status == 200:  login\_data = await login\_response.json()  token = login\_data.get('access\_token')  if token:  response = RedirectResponse(url="/tasks", status\_code=303)  response.set\_cookie(key="access\_token", value=token, httponly=True)  logger.info(f"Set cookie and redirecting to /tasks for {email}")  return response  try:  error\_data = await login\_response.json()  error\_message = error\_data.get('detail', 'Unknown error')  except aiohttp.ContentTypeError:  error\_message = await login\_response.text()  logger.error(f"Auto-login failed: {error\_message}")  return templates.TemplateResponse(  "register.html",  {  "request": request,  "error": f"Auto-login failed: {error\_message}",  "success": None  }  )  else:  try:  error\_data = await reg\_response.json()  error\_message = error\_data.get('detail', 'Unknown error')  except aiohttp.ContentTypeError:  error\_message = await reg\_response.text()  logger.error(f"Registration failed: {error\_message}")  return templates.TemplateResponse(  "register.html",  {  "request": request,  "error": f"Registration failed: {error\_message}",  "success": None  }  )  except aiohttp.ClientError as e:  logger.error(f"Registration error: {str(e)}")  return templates.TemplateResponse(  "register.html",  {  "request": request,  "error": f"Error connecting to server: {str(e)}",  "success": None  }  )  @router.get("/login", response\_class=HTMLResponse) async def login\_page(request: Request):  return templates.TemplateResponse(  "login.html",  {"request": request, "error": None}  )  @router.post("/login", response\_class=HTMLResponse) async def login\_user(  request: Request,  username: str = Form(...),  password: str = Form(...), ):  async with aiohttp.ClientSession() as session:  try:  async with session.post(  'http://localhost:8000/auth/jwt/login',  data={  'username': username,  'password': password  },  headers={'Content-Type': 'application/x-www-form-urlencoded'}  ) as response:  logger.info(f"Login response: {response.status} {await response.text()}, headers: {response.headers}")  if response.status == 200:  response\_data = await response.json()  token = response\_data.get('access\_token')  if token:  response = RedirectResponse(url="/tasks", status\_code=303)  response.set\_cookie(key="access\_token", value=token, httponly=True)  logger.info(f"Set cookie and redirecting to /tasks for {username}")  return response  else:  logger.error("No access\_token in login response")  return templates.TemplateResponse(  "login.html",  {  "request": request,  "error": "Login failed: No access token received",  "success": None  }  )  else:  try:  error\_data = await response.json()  error\_message = error\_data.get('detail', 'Unknown error')  except aiohttp.ContentTypeError:  error\_message = await response.text()  logger.error(f"Login failed: {error\_message}")  return templates.TemplateResponse(  "login.html",  {  "request": request,  "error": f"Login failed: {error\_message}",  "success": None  }  )  except aiohttp.ClientError as e:  logger.error(f"Login error: {str(e)}")  return templates.TemplateResponse(  "login.html",  {  "request": request,  "error": f"Error connecting to server: {str(e)}",  "success": None  }  )  @router.get("/tasks", response\_class=HTMLResponse) async def tasks\_page(request: Request):  token = request.cookies.get("access\_token")  if not token:  logger.info("No token found, redirecting to /login")  return RedirectResponse(url="/login", status\_code=303)   async with aiohttp.ClientSession() as session:  try:  async with session.get(  'http://localhost:8000/tasks/',  headers={'Authorization': f'Bearer {token}'}  ) as response:  logger.info(f"Get tasks response: {response.status} {await response.text()}, headers: {response.headers}")  if response.status == 200:  tasks = await response.json()  return templates.TemplateResponse(  "tasks.html",  {  "request": request,  "tasks": tasks,  "error": None,  "success": None  }  )  elif response.status == 401:  logger.info("Unauthorized, clearing token and redirecting to /login")  response = RedirectResponse(url="/login", status\_code=303)  response.delete\_cookie("access\_token")  return response  else:  try:  error\_data = await response.json()  error\_message = error\_data.get('detail', 'Unknown error')  except aiohttp.ContentTypeError:  error\_message = await response.text()  logger.error(f"Failed to fetch tasks: {error\_message}")  return templates.TemplateResponse(  "tasks.html",  {  "request": request,  "tasks": [],  "error": f"Failed to fetch tasks: {error\_message}",  "success": None  }  )  except aiohttp.ClientError as e:  logger.error(f"Get tasks error: {str(e)}")  return templates.TemplateResponse(  "tasks.html",  {  "request": request,  "tasks": [],  "error": f"Error connecting to server: {str(e)}",  "success": None  }  )  @router.post("/tasks", response\_class=HTMLResponse) async def create\_task(  request: Request,  name: str = Form(...),  text\_of\_task: str = Form(None), ):  token = request.cookies.get("access\_token")  if not token:  logger.info("No token found, redirecting to /login")  return RedirectResponse(url="/login", status\_code=303)   async with aiohttp.ClientSession() as session:  try:  async with session.post(  'http://localhost:8000/tasks/',  json={  'name': name,  'text\_of\_task': text\_of\_task  },  headers={'Authorization': f'Bearer {token}'}  ) as response:  logger.info(f"Create task response: {response.status} {await response.text()}, headers: {response.headers}")  if response.status == 201:  return RedirectResponse(url="/tasks", status\_code=303)  elif response.status == 401:  logger.info("Unauthorized, clearing token and redirecting to /login")  response = RedirectResponse(url="/login", status\_code=303)  response.delete\_cookie("access\_token")  return response  else:  try:  error\_data = await response.json()  error\_message = error\_data.get('detail', 'Unknown error')  except aiohttp.ContentTypeError:  error\_message = await response.text()  logger.error(f"Failed to create task: {error\_message}")  async with session.get(  'http://localhost:8000/tasks/',  headers={'Authorization': f'Bearer {token}'}  ) as tasks\_response:  tasks = await tasks\_response.json() if tasks\_response.status == 200 else []  return templates.TemplateResponse(  "tasks.html",  {  "request": request,  "tasks": tasks,  "error": f"Failed to create task: {error\_message}",  "success": None  }  )  except aiohttp.ClientError as e:  logger.error(f"Create task error: {str(e)}")  async with session.get(  'http://localhost:8000/tasks/',  headers={'Authorization': f'Bearer {token}'}  ) as tasks\_response:  tasks = await tasks\_response.json() if tasks\_response.status == 200 else []  return templates.TemplateResponse(  "tasks.html",  {  "request": request,  "tasks": tasks,  "error": f"Error connecting to server: {str(e)}",  "success": None  }  )  @router.get("/logout", response\_class=HTMLResponse) async def logout(request: Request):  logger.info("Logging out, clearing token")  response = RedirectResponse(url="/")  response.delete\_cookie("access\_token")  return response |

Таблица1 –View функции для шаблонов

|  |
| --- |
| from datetime import datetime from aiogram import Router, F, types from aiogram.filters import CommandStart from aiogram.types import ReplyKeyboardMarkup, KeyboardButton from aiogram.fsm.context import FSMContext from aiogram.fsm.state import State, StatesGroup import aiohttp import logging  logging.basicConfig(level=logging.INFO) logger = logging.getLogger(\_\_name\_\_)  router = Router()  class RegistrationStates(StatesGroup):  waiting\_for\_email = State()  waiting\_for\_password = State()  class LoginStates(StatesGroup):  waiting\_for\_email = State()  waiting\_for\_password = State()  class TaskCreationStates(StatesGroup):  waiting\_for\_name = State()  waiting\_for\_text = State()  # Create keyboards for main menu def get\_main\_menu(is\_authenticated: bool = False) -> ReplyKeyboardMarkup:  if is\_authenticated:  return ReplyKeyboardMarkup(  keyboard=[  [KeyboardButton(text='Получить таски'), KeyboardButton(text='Создать таск')],  [KeyboardButton(text='Выйти')]  ],  resize\_keyboard=True  )  return ReplyKeyboardMarkup(  keyboard=[  [  KeyboardButton(text='Зарегистрироваться'),  KeyboardButton(text='Логин'),  ],  ],  resize\_keyboard=True  )  # Create keyboard for skipping text\_of\_task def get\_skip\_keyboard() -> ReplyKeyboardMarkup:  return ReplyKeyboardMarkup(  keyboard=[  [KeyboardButton(text='Пропустить')]  ],  resize\_keyboard=True  )  # Start command handler @router.message(CommandStart()) async def command\_start(message: types.Message, state: FSMContext) -> None:  user\_data = await state.get\_data()  is\_authenticated = 'access\_token' in user\_data  logger.info(f"Start command: is\_authenticated={is\_authenticated}, user\_data={user\_data}")  await message.answer(  'Hi! What do you want?',  reply\_markup=get\_main\_menu(is\_authenticated),  )  if not is\_authenticated:  await state.clear()  # Registration button handler @router.message(F.text == 'Зарегистрироваться') async def start\_registration(message: types.Message, state: FSMContext) -> None:  user\_data = await state.get\_data()  if 'access\_token' in user\_data:  await message.answer(  "You are already registered and logged in!",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  return  await message.answer(  "Please enter your email:",  reply\_markup=types.ReplyKeyboardRemove()  )  await state.set\_state(RegistrationStates.waiting\_for\_email)  # Handle email input for registration @router.message(RegistrationStates.waiting\_for\_email) async def process\_email(message: types.Message, state: FSMContext) -> None:  if '@' not in message.text:  await message.answer("Please enter a valid email address:")  return  await state.update\_data(email=message.text)  await message.answer("Please enter your password:")  await state.set\_state(RegistrationStates.waiting\_for\_password)  # Handle password input for registration @router.message(RegistrationStates.waiting\_for\_password) async def process\_password(message: types.Message, state: FSMContext) -> None:  await state.update\_data(password=message.text)  user\_data = await state.get\_data()   async with aiohttp.ClientSession() as session:  try:  async with session.post(  'http://localhost:8000/auth/register',  json={  'email': user\_data['email'],  'password': user\_data['password'],  'is\_active': True,  'is\_superuser': False,  'is\_verified': False  }  ) as response:  logger.info(f"Register response: {response.status} {await response.text()}")  if response.status == 201:  response\_data = await response.json()  token = response\_data.get('access\_token')  if token:  await state.update\_data(access\_token=token)  logger.info(f"Stored token: {token}")  await message.answer(  f"Registration successful!\nEmail: {user\_data['email']}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  else:  try:  error\_data = await response.json()  await message.answer(  f"Registration failed: {error\_data.get('detail', 'Unknown error')}",  reply\_markup=get\_main\_menu()  )  except aiohttp.ContentTypeError:  error\_text = await response.text()  await message.answer(  f"Registration failed: Server returned non-JSON response: {error\_text}",  reply\_markup=get\_main\_menu()  )  except aiohttp.ClientError as e:  logger.error(f"Register error: {str(e)}")  await message.answer(  f"Error connecting to server: {str(e)}",  reply\_markup=get\_main\_menu()  )  await state.set\_state(None)  # Login button handler @router.message(F.text == 'Логин') async def start\_login(message: types.Message, state: FSMContext) -> None:  user\_data = await state.get\_data()  if 'access\_token' in user\_data:  await message.answer(  "You are already logged in!",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  return  await message.answer(  "Please enter your email:",  reply\_markup=types.ReplyKeyboardRemove()  )  await state.set\_state(LoginStates.waiting\_for\_email)  # Handle login email input @router.message(LoginStates.waiting\_for\_email) async def process\_login\_email(message: types.Message, state: FSMContext) -> None:  if '@' not in message.text:  await message.answer("Please enter a valid email address:")  return  await state.update\_data(email=message.text)  await message.answer("Please enter your password:")  await state.set\_state(LoginStates.waiting\_for\_password)  # Handle login password input @router.message(LoginStates.waiting\_for\_password) async def process\_login\_password(message: types.Message, state: FSMContext) -> None:  user\_data = await state.get\_data()   async with aiohttp.ClientSession() as session:  try:  async with session.post(  'http://localhost:8000/auth/jwt/login',  data={  'username': user\_data['email'],  'password': message.text  },  headers={'Content-Type': 'application/x-www-form-urlencoded'}  ) as response:  logger.info(f"Login response: {response.status} {await response.text()}")  if response.status == 200:  response\_data = await response.json()  token = response\_data.get('access\_token')  if token:  await state.update\_data(access\_token=token)  logger.info(f"Stored token: {token}")  await message.answer(  f"Login successful!\nEmail: {user\_data['email']}\nYou can now get your tasks!",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  else:  try:  error\_data = await response.json()  await message.answer(  f"Login failed: {error\_data.get('detail', 'Unknown error')}",  reply\_markup=get\_main\_menu()  )  except aiohttp.ContentTypeError:  error\_text = await response.text()  await message.answer(  f"Login failed: Server returned non-JSON response: {error\_text}",  reply\_markup=get\_main\_menu()  )  except aiohttp.ClientError as e:  logger.error(f"Login error: {str(e)}")  await message.answer(  f"Error connecting to server: {str(e)}",  reply\_markup=get\_main\_menu()  )  await state.set\_state(None)  # Logout handler @router.message(F.text == 'Выйти') async def logout(message: types.Message, state: FSMContext) -> None:  await state.clear()  await message.answer(  "You have been logged out.",  reply\_markup=get\_main\_menu(is\_authenticated=False)  )  # Handle "Get Tasks" button @router.message(F.text == 'Получить таски') async def get\_tasks(message: types.Message, state: FSMContext) -> None:  user\_data = await state.get\_data()  if 'access\_token' not in user\_data:  await message.answer(  "Please log in or register first!",  reply\_markup=get\_main\_menu(is\_authenticated=False)  )  return  logger.info(f"Get tasks with token: {user\_data['access\_token']}")   async with aiohttp.ClientSession() as session:  try:  async with session.get(  'http://localhost:8000/tasks/',  headers={'Authorization': f'Bearer {user\_data["access\_token"]}'}  ) as response:  logger.info(f"Get tasks response: {response.status} {await response.text()}")  if response.status == 200:  tasks = await response.json()  if tasks:  tasks\_list = "\n".join([  f"Task {i + 1}:\nname: {task['name']}\ntext: {task.get('text\_of\_task', 'None')}\ncreated\_at: {task['created\_at']}"  for i, task in enumerate(tasks)  ])  await message.answer(  f"Your tasks:\n{tasks\_list}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  else:  await message.answer(  "You have no tasks.",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  elif response.status == 401:  await state.clear()  await message.answer(  "Your session has expired. Please log in again.",  reply\_markup=get\_main\_menu(is\_authenticated=False)  )  else:  try:  error\_data = await response.json()  await message.answer(  f"Failed to fetch tasks: {error\_data.get('detail', 'Unknown error')}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  except aiohttp.ContentTypeError:  error\_text = await response.text()  await message.answer(  f"Failed to fetch tasks: Server returned non-JSON response: {error\_text}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  except aiohttp.ClientError as e:  logger.error(f"Get tasks error: {str(e)}")  await message.answer(  f"Error connecting to server: {str(e)}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  # Handle "Create Task" button @router.message(F.text == 'Создать таск') async def start\_task\_creation(message: types.Message, state: FSMContext) -> None:  user\_data = await state.get\_data()  if 'access\_token' not in user\_data:  await message.answer(  "Please log in or register first!",  reply\_markup=get\_main\_menu(is\_authenticated=False)  )  return  await message.answer(  "Please enter the task name:",  reply\_markup=types.ReplyKeyboardRemove()  )  await state.set\_state(TaskCreationStates.waiting\_for\_name)  # Handle task name input @router.message(TaskCreationStates.waiting\_for\_name) async def process\_task\_name(message: types.Message, state: FSMContext) -> None:  if not message.text.strip():  await message.answer("Task name cannot be empty. Please enter a valid name:")  return  await state.update\_data(task\_name=message.text.strip())  await message.answer(  "Please enter the task description (or press 'Пропустить' to leave it empty):",  reply\_markup=get\_skip\_keyboard()  )  await state.set\_state(TaskCreationStates.waiting\_for\_text)  # Handle task text input or skip @router.message(TaskCreationStates.waiting\_for\_text) async def process\_task\_text(message: types.Message, state: FSMContext) -> None:  user\_data = await state.get\_data()  task\_text = None if message.text == 'Пропустить' else message.text.strip()  logger.info(f"Creating task with token: {user\_data['access\_token']}, name: {user\_data['task\_name']}, text: {task\_text}")   async with aiohttp.ClientSession() as session:  try:  async with session.post(  'http://localhost:8000/tasks/',  json={  'name': user\_data['task\_name'],  'text\_of\_task': task\_text  },  headers={'Authorization': f'Bearer {user\_data["access\_token"]}'}  ) as response:  logger.info(f"Create task response: {response.status} {await response.text()}")  if response.status == 201:  response\_data = await response.json()  await message.answer(  f"Task created successfully!\nname: {response\_data['name']}\ntext: {response\_data.get('text\_of\_task', 'None')}\ncreated\_at: {response\_data['created\_at']}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  elif response.status == 401:  await state.clear()  await message.answer(  "Your session has expired. Please log in again.",  reply\_markup=get\_main\_menu(is\_authenticated=False)  )  else:  try:  error\_data = await response.json()  await message.answer(  f"Failed to create task: {error\_data.get('detail', 'Unknown error')}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  except aiohttp.ContentTypeError:  error\_text = await response.text()  await message.answer(  f"Failed to create task: Server returned non-JSON response: {error\_text}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  except aiohttp.ClientError as e:  logger.error(f"Create task error: {str(e)}")  await message.answer(  f"Error connecting to server: {str(e)}",  reply\_markup=get\_main\_menu(is\_authenticated=True)  )  await state.set\_state(None) # Clear only task creation state, keep access\_token |

Таблица 2 – Хендлеры бота

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.13 – Env файл

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.14 – Запуск бота

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.15 – Запуск FastApi

## **Использование ИИ для работы**

Кинул запрос для создания темплейтов, на что ИИ без проблем дал ответ и рабочие коды. Ниже представлены запрос:

Изображение выглядит как текст, снимок экрана, меню, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.16 – запрос к ИИ

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.17 – запрос к ИИ с получением кодов

# **Работа сервиса и бота**

Изображение выглядит как текст, программное обеспечение, Мультимедийное программное обеспечение, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.18 - Запуск сервиса

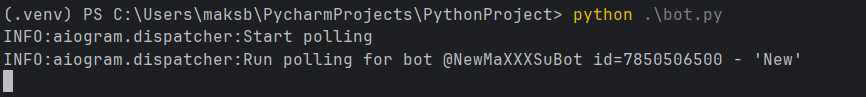


Рис.19 – Запуск бота

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.20 – окно регистрации/входа на сервисе

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рис.21 - окно создания задач с сервиса

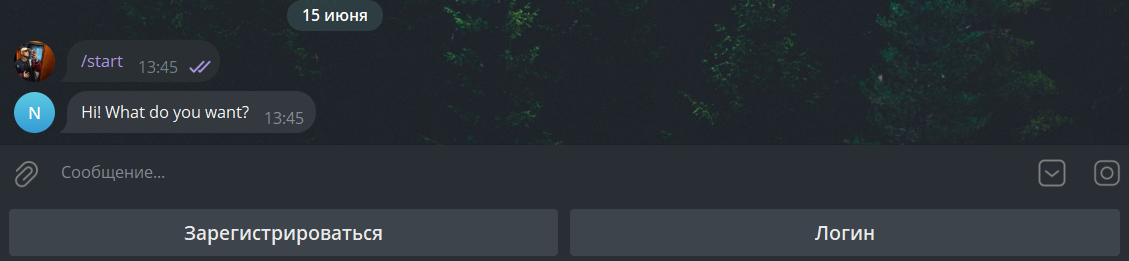
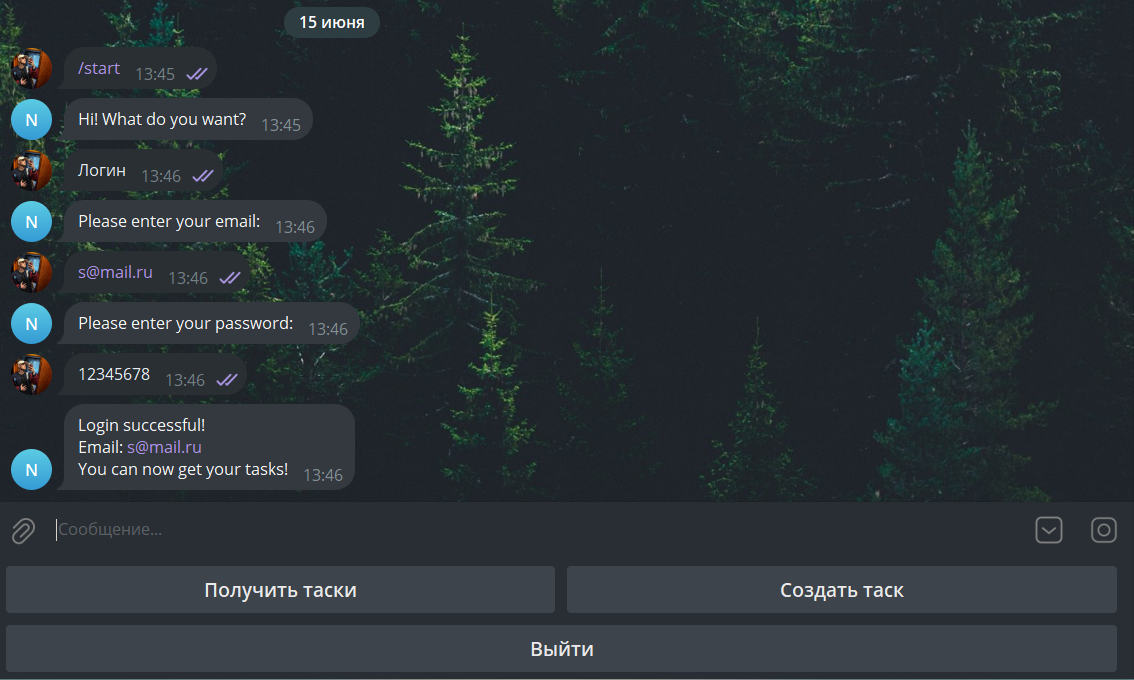


Рис.22 – стартовое меню бота

   
Рис.23 – вход

Изображение выглядит как снимок экрана, текст, Мультимедийное программное обеспечение, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.  
Рис.24 – Меню получения и создания задач

## **Вывод**

Создание Web-Сервиса для регистрации и создания задач, интегрированные с Telegram ботом, использованием ИИ прошло успешно. Изучил фреймворки на Python такие как Fastapi, Alembic и тд. Бот успешно выполняет свои задачи. Также и ИИ успешно справилась со своей задачей.