УДК 004.423

НАЧАЛО РАБОТЫ С FASTAPI. ПРОСТЕЙШАЯ СИСТЕМА АВТОРИЗАЦИИ И АУТЕНТИФИКАЦИИ ПОЛЬЗОВАТЕЛЯ

Халевин Т.А.

студент направления подготовки информатики и вычислительной техники,

Хакасский государственный университет имени Н.Ф. Катанова,

г. Абакан, Россия [1]

Аннотация: В данной статье излагается методика создания системы аутентификации и регистрации пользователей в веб-приложениях, используя фреймворк FastAPI. Обсуждаются ключевые аспекты безопасного хранения паролей, генерации токенов доступа с использованием JWT и внедрения этих механизмов в процесс взаимодействия с пользователем. Через примеры кода с пояснениями демонстрируется, как на практике реализовать регистрацию новых пользователей, их аутентификацию и авторизацию для доступа к защищенным ресурсам приложения.

Ключевые слова: Python, FastAPI, аутентификация, регистрация пользователей, JWT, безопасность, веб-приложение, хеширование паролей.

GETTING STARTED WITH FASTAPI. THE SIMPLEST SYSTEM OF USER AUTHORISATION AND AUTHENTICATION.

Khalevin T.A.

student of computer science and computer engineering department,

N.F. Katanov Khakass State University,

Abakan, Russia

[1] Научный руководитель: Голубничий А.А. старший преподаватель кафедры ПОВТиАС, Хакасский государственный университет имени Н.Ф. Катанова, г. Абакан, Россия

Abstract: This paper presents a methodology for creating a user authentication and registration system for web applications using the FastAPI framework. It discusses key aspects of secure password storage, access token generation using JWT, and the implementation of these mechanisms in the user interaction process. Code examples with explanations are used to demonstrate how to practically implement registration of new users, their authentication and authorisation to access secure application resources.

Keywords: Python, FastAPI, authentication, user registration, JWT, security, web application, password hashing.

FastAPI является современным, асинхронным веб-фреймворком для создания API на языке программирования Python, ориентированным на быструю разработку, легкость в использовании и высокую производительность. Разработанный Тьяго Карнейро, фреймворк быстро набрал популярность среди разработчиков благодаря своей способности значительно ускорять процесс разработки без потери качества и эффективности.

Основываясь на стандартных типах Python, FastAPI автоматически генерирует документацию для API (с использованием Swagger UI и ReDoc), предоставляет типизированные ответы и запросы, что значительно уменьшает количество ошибок и упрощает тестирование. FastAPI интегрируется с Pydantic для валидации данных, что позволяет использовать Python type hints для валидации входящих запросов, а также для сериализации ответов.

Основные преимущества FastAPI:

- FastAPI является одним из самых быстрых веб-фреймворков для Python, сопоставимым по скорости с NodeJS и Go благодаря Starlette и Pydantic, лежащим в его основе.
- Сочетание Python type hints и автоматической документации делает FastAPI очень легким для изучения и использования. Разработчики

могут сосредоточиться на логике своего приложения, а не на написании дополнительного кода для документации или валидации данных.

- FastAPI полностью асинхронен и позволяет использовать async и await для асинхронного программирования, что делает его идеальным для высоконагруженных приложений и микросервисов.
- Фреймворк автоматически генерирует документацию для API (с помощью Swagger UI и ReDoc), что облегчает тестирование и взаимодействие с API как для разработчиков, так и для конечных пользователей.
- FastAPI предоставляет множество встроенных решений для аутентификации, авторизации и безопасности, включая поддержку OAuth2 с JWT tokens, базовую аутентификацию и другие.
- Благодаря поддержке асинхронных и синхронных обработчиков, а также возможности легкой интеграции с другими асинхронными библиотеками и системами, FastAPI может быть адаптирован под любые требования проекта.
- FastAPI можно использовать в сочетании с любыми ORM, базами данных и сторонними библиотеками, что делает его универсальным инструментом для создания различных веб-приложений.

Для начала работы с FastAPI необходимо установить сам фреймворк и ASGI-сервер, например, Uvicorn, который позволит запускать наше приложение. Для установки выполните следующую команду: pip install fastapi uvicorn.

После этого можно создать первое простейшее API для демонстрации. Для начала создадим файл main.py. Пример данного файла представлен на рисунке 1.

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def read_root():
    return {"Hello": "World"}
```

Рисунок 1 – Пример простейшего маршрута [разработано автором]

В примере создается экземпляр FastAPI и определяем корневой маршрут с помощью декоратора @app.get("/"). Функция read_root асинхронна, что позволяет обрабатывать запросы эффективно, и возвращает простой JSON-объект [1].

Теперь при запуске проекта командой python -m uvicorn main:app –reload, флаг –reload заставляет сервер перезапускаться при изменении кода, что очень удобно во время разработки. На главной странице API мы получим JSON с ключом Hello и значением World. Главная страница представлена на рисунке 2.

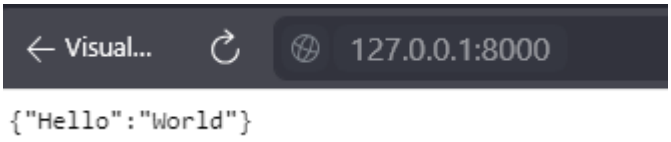

Рисунок 2 – Главная страница [разработано автором]

FastAPI делает легким процесс создания API с различными маршрутами. Добавим еще один маршрут в наш код для демонстрации. Он представлен на рисунке 3.

```
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```

Рисунок 3 – Дополнительный маршрут [разработано автором]

Данный маршрут использует параметр пути item_id, который автоматически валидируется как целое число благодаря аннотации типа int [2].

Теперь, если перейти по данному маршруту, мы будем получать JSON в котором в качестве ключа будет item_id, а в качестве значения целое число из

маршрута по которому мы переходим. На рисунке 4 показан результат перехода по маршруту http://127.0.0.1:8000/items/123.

Рисунок 4 – Отображение данных в браузере при переходе по дополнительному маршруту [разработано автором]

FastAPI предоставляет разработчикам мощные инструменты для реализации аутентификации и авторизации. В отличие от Flask, FastAPI использует современные подходы, такие как OAuth2 с JWT (JSON Web Tokens) для создания безопасных и масштабируемых приложений. Давайте рассмотрим, как можно реализовать систему аутентификации на примере использования JWT. Для работы с JWT в FastAPI требуется установить дополнительные библиотеки. Выполните следующую команду для их установки: pip install "pydantic[email]" python-jose[cryptography] passlib[bcrypt]. Эти пакеты позволят вам создавать и обрабатывать JWT токены, валидировать пользовательские данные и безопасно хранить пароли.

Первым шагом будет создание Pydantic моделей для пользовательских данных. Модели Pydantic используются для валидации и конвертации входящих данных в JSON формате в Python объекты [3].

Сначала необходимо создать файл models.py, после чего в нем будет реализован код представленный на рисунке 5.

```
from pydantic import BaseModel, EmailStr

class UserBase(BaseModel):
    email: EmailStr

class UserInDB(UserBase):
    hashed_password: str

class UserCreate(UserBase):
    password: str

class Token(BaseModel):
    access_token: str
    token_type: str
```

Рисунок 5 – Определение моделей данных с использованием Pydantic [разработано автором]

Фрагмент кода выше демонстрирует использование Pydantic, библиотеки для валидации и управления данными в Python, для создания моделей данных, которые могут быть использованы в приложении на FastAPI для аутентификации и авторизации пользователей, а также для управления токенами доступа.

UserBase является базовым классом для моделей пользователя, определяющим поле email. BaseModel из Pydantic используется как базовый класс, предоставляющий функционал валидации данных. EmailStr — это специальный тип Pydantic, который валидирует строки как адреса электронной почты. Значит любое значение, присвоенное email в экземпляре UserBase или его потомках, должно быть корректным адресом электронной почты, иначе Pydantic выдаст ошибку валидации.

UserInDB расширяет UserBase, добавляя поле hashed_password. Данный класс предназначен для использования с данными пользователя, хранящимися в базе данных. В отличие от простой строки password, hashed_password содержит хешированный пароль пользователя, что является хорошей практикой безопасности.

UserCreate также наследуется от UserBase, но вместо хешированного пароля содержит обычную строку password. Этот класс может быть использован при регистрации нового пользователя, когда пользователь отправляет свой пароль в открытом виде. После получения такого пароля сервер должен его хешировать перед сохранением в базу данных как hashed_password в экземпляре UserInDB.

Token – это модель данных для JWT (JSON Web Tokens), которые используются для аутентификации и авторизации в веб-приложениях. access_token содержит сам токен, который клиент должен предоставить для доступа к защищенным ресурсам. token_type обычно содержит тип токена, который обычно является "bearer", указывая на то, что токен должен быть предоставлен в заголовке авторизации HTTP-запроса.

Теперь реализуем main.py. В нем будет содержаться вся логика нашего приложения. На рисунке 6 представлены необходимые импорты.

```
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from datetime import datetime, timedelta, timezone
from passlib.context import CryptContext
from typing import Union

from models import UserInDB, UserCreate, Token
```

Рисунок 6 – Необходимые импорты [разработано автором]

FastAPI – основной фреймворк для создания веб-приложения и API.

Depends – используется для внедрения зависимостей, таких как функции аутентификации.

HTTPException – позволяет выбрасывать исключения HTTP с определенным статус-кодом.

status – содержит статус-коды HTTP для использования в исключениях.

OAuth2PasswordBearer и OAuth2PasswordRequestForm – инструменты для работы с OAuth2, облегчающие процесс аутентификации через пароли.

jwt (из библиотеки jose) – используется для создания и декодирования JWT токенов.

datetime, timedelta, timezone – для работы с датами и временем, необходимы при установке срока действия токенов.

CryptContext (из passlib) – предоставляет инструменты для хеширования и проверки паролей.

Union (из typing) – для указания типов данных, которые могут быть одним из нескольких типов (используется в определении времени истечения токена).

Модели (UserInDB, UserCreate, Token) – пользовательские классы, определенные в модуле models, для работы с данными пользователей и токенов.

В качестве базы данных будет использоваться словарь для наглядности. В реальном приложении ни в коем случае не нужно использовать такой же подход [4].

На рисунке 7 представлен словарь выполняющий роль базы данных в приложении.

```
# Фиктивная "база данных"
fake_users_db = {
    "email@example.com": {
        "email": "email@example.com",
        "hashed_password": "$2b$12$K4ocY.uI5wxH9.bJP6z7EOCj7UhW3/hATCI.TgSXeXK1v.IQFHyE6"
    }
}
```

Рисунок 7 – Словарь выполняющий функцию БД [разработано автором]

Теперь опишем вспомогательные функции для нашего приложения. Они представлены на рисунке 8.

```
# Конфигурация
SECRET_KEY = "09d25e094faa6ca2556c818166b7a9563b93f7099f6f0f4caa6cf63b88e8d3e7"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

app = FastAPI()

def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)

def get_user(db, email: str):
    if email in db:
        user_dict = db[email]
        return UserInDB(**user_dict)

def authenticate_user(fake_db, email: str, password: str):
    user = get_user(fake_db, email)
    if not user or not verify_password(password, user.hashed_password):
        return False
    return user

def create_access_token(data: dict, expires_delta: Union[timedelta, None] = None):
    to_encode = data.copy()
    if expires_delta:
        expire = datetime.now(timezone.utc) + expires_delta
    else:
        expire = datetime.now(timezone.utc) + timedelta(minutes=15)
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

Рисунок 8 – Вспомогательные функции приложения [разработано автором]

Конфигурация (SECRET_KEY, ALGORITHM, ACCESS_TOKEN_ EXPIRE_MINUTES): Настройки для создания JWT токенов.

Вспомогательные функции:

- verify_password и get_password_hash для работы с паролями.
- get_user и authenticate_user для аутентификации пользователя.
- create_access_token для создания JWT токенов.

Теперь рассмотрим маршруты необходимые для регистрации и авторизации пользователя, а также маршрут доступный только авторизированному пользователю. Маршруты приложения представлены на рисунке 9.

```
@app.post("/token", response_model=Token)
async def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends()):
    user = authenticate_user(fake_users_db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.email}, expires_delta=access_token_expires
    )
    return {"access_token": access_token, "token_type": "bearer"}

@app.get("/users/me")
async def read_users_me(token: str = Depends(oauth2_scheme)):
    return {"token": token}

@app.post("/register", response_model=UserInDB)
async def register_user(user: UserCreate):
    if user.email in fake_users_db:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Email already registered"
        )
    hashed_password = get_password_hash(user.password)
    user_dict = user.model_dump()
    user_dict['hashed_password'] = hashed_password
    del user_dict['password']  # Удаляем исходный пароль из словаря
    fake_users_db[user.email] = user_dict
    return UserInDB(**user_dict)
```

Рисунок 9 – Маршруты приложения [разработано автором]

Маршрут для получения токена /token служит для аутентификации пользователя и выдачи JWT, который в дальнейшем используется для доступа к защищенным маршрутам.

Метод и возвращаемый тип: Используется декоратор @app.post("/token", response_model=Token) для указания, что этот маршрут принимает POST-запросы и возвращает объект типа Token.

Аргументы функции: form_data: OAuth2PasswordRequestForm = Depends() – OAuth2PasswordRequestForm автоматически извлекает данные из формы запроса (username и password). Использование Depends() позволяет внедрить выполнение дополнительной логики, в данном случае извлечение данных формы.

Сначала вызывается функция authenticate_user, чтобы проверить, существует ли пользователь с таким email и паролем в "базе данных" и верен ли пароль. Если пользователь не найден или пароль неверен, генерируется исключение HTTPException со статусом 401 (Неавторизовано).

Если аутентификация прошла успешно, создается JWT с помощью функции create_access_token, в который включается информация о пользователе (sub с email пользователя) и срок его действия. Токен возвращается клиенту в ответе.

Маршрут для получения данных текущего пользователя /users/me позволяет получить данные текущего аутентифицированного пользователя, используя предоставленный JWT. Принимает GET-запросы, возвращает данные о токене, но в реальном приложении здесь можно возвращать более конкретную информацию о пользователе.

token: str = Depends(oauth2_scheme) – oauth2_scheme извлекает токен из заголовка Authorization запроса. Это встроенная зависимость FastAPI для работы с OAuth2.

В данной реализации функция просто возвращает токен. В реальном приложении здесь следует проверить валидность токена и, если он валиден, вернуть информацию о пользователе, связанную с этим токеном.

Маршрут для регистрации нового пользователя /register используется для регистрации новых пользователей в системе.

Обрабатывает POST-запросы и возвращает объект типа UserInDB, содержащий информацию о зарегистрированном пользователе.

user: UserCreate – Pydantic модель, которая валидирует и представляет данные пользователя, переданные в запросе.

Сначала проверяется, не зарегистрирован ли уже пользователь с таким email. Если зарегистрирован, генерируется исключение HTTPException со статусом 400 (Неверный запрос). Если нет, пароль хешируется, создается новый объект UserInDB с хешированным паролем и email пользователя, и добавляется в "базу данных".

В ответ клиенту возвращается информация о только что созданном пользователе (без пароля).

Эти маршруты вместе образуют основу для системы аутентификации и регистрации в приложении FastAPI, используя JWT для безопасной работы с API.

Теперь при запуске приложения и переходе по адресу http://127.0.0.1:8000/docs/ будет доступна документация, автоматически сгенерированная FastAPI. Для проверки нашего приложения сначала попробуйте выполнить маршрут /users/me. Вы получите ошибку. Для того чтобы получить доступ к данному маршруту:

1. Зарегистрируйте пользователя через маршрут /register.

2. После этого в правом верхнем углу авторизуйтесь по созданному пользователю

3. Выполните маршрут /users/me.

Вы получите ответ с токеном вашего пользователя. Последовательность представлена на рисунке 10.

Рисунок 10 – Последовательность для проверки приложения [разработано автором]

В заключение, представленный код демонстрирует основы создания системы аутентификации и регистрации пользователей в веб-приложении на основе фреймворка FastAPI. Использование FastAPI в сочетании с такими инструментами, как Pydantic для валидации данных и Passlib для безопасного хеширования паролей, позволяет эффективно и безопасно управлять пользовательскими данными. JWT токены добавляют дополнительный уровень безопасности, обеспечивая аутентификацию и авторизацию пользователей при работе с защищенными маршрутами API.

Хотя в данном примере использовалась фиктивная "база данных" для упрощения демонстрации, в реальных проектах рекомендуется применять настоящие базы данных для хранения пользовательских данных. Также важно принимать во внимание дополнительные меры безопасности, такие как обновление и отзыв токенов, защита от различных видов атак (например, CSRF или XSS) и использование HTTPS для шифрования трафика между клиентом и сервером.

FastAPI предоставляет мощные и гибкие инструменты для разработки современных веб-приложений с поддержкой асинхронного программирования, автоматической документации API и встроенной поддержкой аутентификации. Благодаря своей производительности, легкости в использовании и широкому спектру функциональных возможностей,

FastAPI является отличным выбором для разработчиков, стремящихся создавать быстрые, надежные и безопасные веб-приложения на Python.

Библиографический список:

1. Доусон М. Программируем на Python [Текст] / Доусон М. – СПб.: Издательский Дом ПИТЕР, 2022. – 416 с.
2. Гэддис Т. Начинаем программировать на Python [Текст] / Т. Гэддис. – СПб.: БХВ-Петербург, 2021. – 768 с.
3. Васильев А.Н. Программирование на Python в примерах и задачах [Текст] / Васильев А.Н. – М.: Бомбора, 2021, 2022.
4. Гуриков, С.Р. Основы алгоритмизации и программирования на Python [Текст] / С.Р. Гуриков. – М.: ИНФРА-М, 2023. – 343 с.

Оригинальность 89%