

УДК 004.423

ПАНЕЛЬ АДМИНИСТРАТОРА В FASTAPI: СОВРЕМЕННЫЙ ПОДХОД К РАЗРАБОТКЕ ВЕБ-ИНТЕРФЕЙСОВ

Дрянкова Д.А.

*студент направления подготовки информатика и вычислительная техника,
Хакасский государственный университет имени Н.Ф. Катанова,
г. Абакан, Россия ¹*

Аннотация: Актуальность создания эффективных административных панелей растет с усложнением веб-приложений. FastAPI, благодаря асинхронной архитектуре и строгой типизации, представляет собой современную альтернативу традиционным фреймворкам для решения этой задачи. Цель статьи – продемонстрировать комплексный подход к разработке панели администратора на FastAPI, охватывая архитектурные паттерны, безопасность и интеграцию с фронтендом. В результате предложена структура приложения, реализованы системы аутентификации, CRUD-операций и разграничения прав доступа, а также проведен сравнительный анализ с Django. Значимость работы заключается в том, что представленные подходы позволяют создавать высокопроизводительные, масштабируемые и безопасные административные интерфейсы, соответствующие современным требованиям.

Ключевые слова: FastAPI, панель администратора, веб-разработка, асинхронное программирование, REST API, Python

ADMINISTRATOR PANEL IN FASTAPI: A MODERN APPROACH TO WEB INTERFACE DEVELOPMENT

Dryankova D.A.

*student of computer science and computer engineering department,
N.F. Katanov Khakass State University,*

¹ Научный руководитель: Козлитин Р.А. канд. физ.-мат. наук, доцент кафедры ПОВТиАС, Хакасский государственный университет имени Н.Ф. Катанова, г. Абакан, Россия

Abakan, Russia

Abstract: The relevance of creating effective admin panels is growing as web applications become more complex. FastAPI, thanks to its asynchronous architecture and strict typing, is a modern alternative to traditional frameworks for solving this problem. The purpose of this article is to demonstrate a comprehensive approach to developing an administrator panel on FastAPI, covering architectural patterns, security, and front-end integration. As a result, an application structure is proposed, authentication systems, CRUD operations, and access rights differentiation are implemented, and a comparative analysis with Django is conducted. The significance of this work lies in the fact that the approaches presented allow for the creation of high-performance, scalable, and secure administrative interfaces that meet modern requirements.

Keywords: FastAPI, admin panel, web development, asynchronous programming, REST API, Python

Введение

Современная веб-разработка требует создания эффективных и масштабируемых решений для управления данными и административных функций. Панели администратора являются критически важным компонентом большинства веб-приложений, обеспечивая интерфейс для управления контентом, пользователями и системными настройками.

FastAPI, появившийся в 2018 году, представляет собой современный высокопроизводительный веб-фреймворк для создания API на языке Python. Основанный на стандартных Python type hints, он обеспечивает автоматическую валидацию данных, генерацию документации и высокую производительность благодаря асинхронной архитектуре.

FastAPI построен на основе нескольких ключевых архитектурных принципов:

- Асинхронность: Использование `async/await` позволяет обрабатывать множественные запросы одновременно без блокировки потока выполнения. Это особенно критично для административных панелей, где необходимо обрабатывать множественные операции с базой данных.

- Type Hints: Использование аннотаций типов Python обеспечивает автоматическую валидацию входных данных и генерацию документации API. Это снижает количество ошибок и упрощает разработку.

- Стандарты OpenAPI: Автоматическая генерация документации в соответствии со стандартами OpenAPI 3.0 обеспечивает интероперабельность и простоту интеграции.

При разработке панелей администратора в FastAPI применяются следующие архитектурные паттерны:

- Repository Pattern: Инкапсулирует логику доступа к данным, обеспечивая гибкость при смене источников данных.

- Dependency Injection: Встроенная система внедрения зависимостей упрощает тестирование и поддержку кода.

- Middleware Pattern: Позволяет обрабатывать запросы на различных уровнях, обеспечивая функциональность аутентификации, логирования и обработки ошибок [1].

Цель и задачи исследования

Цель работы — продемонстрировать современный подход к созданию масштабируемой и безопасной панели администратора с использованием фреймворка FastAPI.

Для достижения поставленной цели были решены следующие задачи:

1. Спроектировать базовую структуру приложения с использованием асинхронных эндпоинтов и системы зависимостей.

2. Реализовать систему аутентификации и авторизации на основе JWT-токенов.

3. Продемонстрировать управление данными с использованием паттерна Repository.
4. Обеспечить безопасность через разграничение прав доступа и систему логирования.
5. Показать способы интеграции с фронтендом, включая REST API и WebSocket.
6. Обеспечить масштабируемость за счет модульности и кэширования.

Базовая структура приложения

На рисунке 1 рассмотрим структуру административной панели на FastAPI.

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import HTTPBearer
from pydantic import BaseModel
from typing import List, Optional

app = FastAPI(title="Admin Panel API", version="1.0.0")
security = HTTPBearer()

# Модель пользователя
class User(BaseModel):
    id: Optional[int] = None
    username: str
    email: str
    is_admin: bool = False

# Dependency для проверки администратора
async def get_admin_user(token: str = Depends(security)):
    # Логика проверки токена и прав администратора
    if not verify_admin_token(token.credentials):
        raise HTTPException(status_code=403, detail="Access denied")
    return get_current_user(token.credentials)
```

Рисунок 1 – Базовая структура приложения

После импорта необходимых модулей, создается экземпляр FastAPI приложения с метаданными и настраивается система безопасности на основе Bearer токенов. После, создаётся модель пользователя, для валидации входящих данных и сериализации ответов. Асинхронная функция проверки прав администратора использует систему зависимостей FastAPI для автоматической инъекции токена безопасности и валидации доступа.

Система аутентификации и авторизации

Безопасность является критически важным компонентом административной панели, рассмотрим реализацию на рисунке 2.

```
from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import jwt

class AuthManager:
    def __init__(self, secret_key: str):
        self.secret_key = secret_key

    async def verify_admin_access(
        self,
        credentials: HTTPAuthorizationCredentials = Depends(HTTPBearer())
    ):
        try:
            payload = jwt.decode(
                credentials.credentials,
                self.secret_key,
                algorithms=["HS256"]
            )
            if not payload.get("is_admin"):
                raise HTTPException(
                    status_code=status.HTTP_403_FORBIDDEN,
                    detail="Insufficient permissions"
                )
            return payload
        except jwt.PyJWTError:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Invalid token"
            )

auth_manager = AuthManager("your-secret-key")

@app.get("/admin/users")
async def get_users(admin_user = Depends(auth_manager.verify_admin_access)):
    return await user_repository.get_all_users()
```

Рисунок 2 – Система аутентификации и авторизации

Для начала подключаются модули для работы с HTTP авторизацией, обработки исключений и JWT токенов. Конструктор класса принимает секретный ключ для подписи и верификации JWT токенов. Асинхронный метод декодирует JWT токен, проверяет административные права и возвращает данные пользователя или генерирует исключения при ошибках. Создается экземпляр менеджера с секретным ключом для всего приложения. Декоратор маршрута использует dependency injection для автоматической проверки прав доступа перед выполнением функции [2].

Управление данными

На рисунке 3, рассмотрим реализацию CRUD-операций с использованием паттерна Repository.

```
from abc import ABC, abstractmethod
from typing import List, Optional

class UserRepository(ABC):
    @abstractmethod
    async def create(self, user_data: User) -> User:
        pass

    @abstractmethod
    async def get_by_id(self, user_id: int) -> Optional[User]:
        pass

    @abstractmethod
    async def update(self, user_id: int, user_data: User) -> User:
        pass

    @abstractmethod
    async def delete(self, user_id: int) -> bool:
        pass

class DatabaseUserRepository(UserRepository):
    async def create(self, user_data: User) -> User:
        # Логика создания пользователя в базе данных
        return user_data

    async def get_by_id(self, user_id: int) -> Optional[User]:
        # Логика получения пользователя из базы данных
        return None

# Административные маршруты
@app.post("/admin/users", response_model=User)
async def create_user(
    user_data: User,
    admin_user = Depends(auth_manager.verify_admin_access),
    repository: UserRepository = Depends(get_user_repository)
):
    return await repository.create(user_data)
```

Рисунок 3 – Управление данными

Сначала определяется абстрактный класс `UserRepository` с четырьмя асинхронными методами (создание, получение, обновление, удаление пользователя) – это интерфейс, который задает контракт для всех реализаций. Затем показывается конкретная реализация `DatabaseUserRepository`, которая

наследует абстрактный класс и реализует его методы для работы с конкретной базой данных. Такая структура позволяет легко менять источник данных (например, с SQL на NoSQL) без изменения бизнес-логики. В конце показан административный маршрут, который использует *dependency injection* для внедрения как проверки прав доступа, так и экземпляра репозитория, что обеспечивает слабую связанность компонентов и упрощает тестирование [3].

Интеграция с фронтенд-решениями

FastAPI обеспечивает создание RESTful API, которые легко интегрируются с современными фронтенд-фреймворками. Рассмотрим пример на рисунке 4.

```
@app.get("/admin/dashboard/stats")
async def get_dashboard_stats(admin_user = Depends(auth_manager.verify_admin_access)):
    return {
        "total_users": await user_repository.count(),
        "active_sessions": await session_repository.count_active(),
        "recent_activities": await activity_repository.get_recent(limit=10)
    }
```

Рисунок 4 – Интеграция с RESTful API

Данный маршрут демонстрирует асинхронное получение различных метрик системы.

Для *real-time* обновлений в административной панели, используется WebSocket соединение, оно представлено на рисунке 5.

```
from fastapi import WebSocket

@app.websocket("/admin/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        # Отправка обновлений в real-time
        stats = await get_real_time_stats()
        await websocket.send_json(stats)
        await asyncio.sleep(5)
```

Рисунок 5 – WebSocket соединение

Данное WebSocket соединение принимает подключение клиента, затем в бесконечном цикле получает актуальную статистику и отправляет её клиенту каждые 5 секунд.

Обеспечение безопасности административной панели

Система разграничения прав доступа на основе разрешений обеспечивает гранулярный контроль над функциональностью административной панели, реализация представлена на рисунке 6.

```
from enum import Enum
import asyncio

class Permission(Enum):
    READ_USERS = "read_users"
    WRITE_USERS = "write_users"
    DELETE_USERS = "delete_users"

def require_permission(permission: Permission):
    def decorator(func):
        async def wrapper(*args, admin_user=Depends(auth_manager.verify_admin_access), **kwargs):
            if permission.value not in admin_user.get("permissions", []):
                raise HTTPException(status_code=403, detail="Insufficient permissions")
            return await func(*args, **kwargs)
        return wrapper
    return decorator

@app.delete("/admin/users/{user_id}")
@require_permission(Permission.DELETE_USERS)
async def delete_user(user_id: int):
    return await user_repository.delete(user_id)
```

Рисунок 6 – Перечисление разрешений

Перечисление определяет доступные разрешения в системе. Фабричная функция создает декоратор для проверки конкретного разрешения. Внутренняя функция-обертка проверяет наличие требуемого разрешения у пользователя перед выполнением защищенной функции. Маршрут удаления пользователя защищен декоратором, который проверяет наличие разрешения DELETE_USERS у текущего администратора.

Система логирования обеспечивает отслеживание действий администраторов для безопасности и соблюдения требований аудита. Данная система представлена на рисунке 7.

```
import logging
from functools import wraps

def audit_log(action: str):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, admin_user=None, **kwargs):
            logging.info(f"Admin {admin_user['username']} performed {action}")
            result = await func(*args, admin_user=admin_user, **kwargs)
            logging.info(f"Action {action} completed successfully")
            return result
        return wrapper
    return decorator
```

Рисунок 7 – Система логирования

Декоратор `audit_log` записывает информацию о выполняемом действии до и после его выполнения.

Масштабируемость и поддержка

На рисунке 8, разделение функциональности на модули обеспечивает масштабируемость.

```
# admin/users/router.py
from fastapi import APIRouter

users_router = APIRouter(prefix="/admin/users", tags=["Admin Users"])

@users_router.get("/")
async def list_users():
    pass

# admin/settings/router.py
settings_router = APIRouter(prefix="/admin/settings", tags=["Admin Settings"])

# Главное приложение
app.include_router(users_router)
app.include_router(settings_router)
```

Рисунок 8 – Разделение роутеров

Сначала создается отдельный роутер для функциональности управления пользователями с префиксом URL и тегами для группировки в документации. После создаётся роутер для настроек системы.

Все модульные роутеры подключаются к основному экземпляру FastAPI приложения, создавая единую точку входа с модульной внутренней структурой.

Использование кэширования для повышения производительности представлено на рисунке 9.

```
from functools import lru_cache
import redis

redis_client = redis.Redis()

@lru_cache(maxsize=128)
async def get_user_stats():
    # Дорогостоящие вычисления статистики
    return await calculate_stats()
```

Рисунок 9 – Кэширование

Подключаются встроенный декоратор LRU-кэша Python и клиент Redis для более сложного кэширования. Декоратор `lru_cache` кэширует результаты функции в памяти приложения. Параметр `maxsize` ограничивает количество сохраняемых результатов, предотвращая чрезмерное потребление памяти [4].

Архитектурные преимущества

Асинхронная архитектура FastAPI обеспечивает высокую производительность административных операций. Благодаря неблокирующим операциям система может обрабатывать множество запросов одновременно. Это также ведет к эффективному использованию ресурсов и меньшему потреблению памяти по сравнению с синхронными фреймворками. В итоге это обеспечивает высокую масштабируемость и способность обрабатывать большое количество одновременных подключений.

Генерация интерактивной документации предоставляет разработчикам мощные инструменты. В их числе – автоматически генерируемый интерфейс для тестирования API (Swagger UI), альтернативная документация с улучшенным дизайном (ReDoc) и стандартизированное описание API для интеграции в виде OpenAPI Schema. Использование строгой типизации повышает надежность и качество кода. Оно обеспечивает автоматическую проверку (валидацию) входящих данных, улучшенную поддержку в средах разработки (IDE Support) и снижение количества ошибок за счет выявления несоответствий типов еще на этапе разработки.

Таблица 1 представлена для наглядного сравнения фреймворков FastAPI и Django REST Framework.

Таблица 1 – Сравнительная таблица FastAPI и Django.

Метрика	FastAPI	Django + DRF
Производительность	~20,000-25,000 (req/sec)	~2,000-3,000 (req/sec)
Асинхронность	Нативная поддержка async/await	Частичная (с Django 3.1+)
Валидация данных	Pydantic (автоматическая)	Serializers (ручная настройка)
Admin панель	Требуется сторонних решений	Встроенная Django Admin
ORM	Поддержка SQLAlchemy, Tortoise	Встроенная Django ORM
Типизация	Строгая (Type Hints обязательны)	Опциональная
Потребление памяти	~50-70 MB	~150-200 MB
Использование	API-first приложения, микросервисы	Full-stack, монолитные приложения

Заключение

FastAPI представляет собой эволюционный шаг в развитии веб-фреймворков Python, объединяя производительность современных асинхронных решений с простотой разработки и поддержки. Как показывают результаты сравнительного анализа, он значительно превосходит традиционные решения в производительности и эффективности использования ресурсов, что делает его идеальным выбором для создания высоконагруженных административных панелей и API-first приложений.

Библиографический список

1. Peralta J. H. Microservice APIs: Using Python, Flask, FastAPI, OpenAPI and More. / Peralta J. H. – Simon and Schuster – 2023.
2. Lathkar M. High-Performance Web Apps with FastAPI / Lathkar M. - California: Apress Berkeley. – 2023.
3. De Luca G. FastAPI Cookbook: Develop high-performance APIs and web applications with Python. / De Luca G. – Packt Publishing Ltd – 2024.
4. Voron F. Building Data Science Applications with FastAPI. / Voron F. – Packt Publishing – 2021.

Оригинальность 75%