# *МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ И КОМПЬЮТЕРНЫХ СЕТЕЙ*

УДК 004.42

## ПОСТРОЕНИЕ ЭФФЕКТИВНОЙ АРХИТЕКТУРЫ ДЛЯ ВЫСОКОНАГРУЖЕННЫХ И МАСШТАБИРУЕМЫХ СЕРВИСОВ НА FASTAPI: ПРИНЦИПЫ И ЛУЧШИЕ ПРАКТИКИ

**К. А. Сермягин,** студент РГРТУ, Рязань, Россия;
orcid.org/0009-0002-6078-4655, e-mail: sermyagin.abcen7@gmail.com.
**А. Н. Пылькин,** д.т.н., профессор кафедры ВПМ РГРТУ, Рязань, Россия;
orcid.org/0000-0001-9925-2870, e-mail: pylkin.a.n@rsreu.ru

*В эпоху роста интернет-сервисов и приложений с высокой нагрузкой разработчики сталкиваются с необходимостью создавать системы, которые могут выдерживать большие объёмы данных и пользователей. Ключевым аспектом успешного проектирования таких систем является выбор подходящей архитектуры и инструментов. FastAPI выделяется среди других технологий своей скоростью, удобством использования и поддержкой современных стандартов.* ***Целью работы*** *является повышение эффективности разработки в контексте фреймворка FastAPI при помощи разработки слоистой архитектуры сервиса.*

***Ключевые слова:*** *FastAPI, веб-фреймворк, API, микросервисы, проектирование, архитектура.*

**DOI:** 10.21667/1995-4565-2025-92-49-56

### Введение

На текущий момент Python быстро развивается и является одним из первых языков программирования по количеству open-source фреймворков. Среди них самыми популярными и востребованными на рынке труда являются: Django, FastAPI, Flask. Django не подходит под написание легковесных микросервисов, однако имеет широкий набор встроенных инструментов. Flask на текущий момент применяется редко по сравнению с FastAPI и Django из-за ограниченных возможностей для масштабирования. FastAPI – это современный веб-фреймворк для Python, который благодаря своей скорости и удобству использования становится всё более популярным среди разработчиков. Быстрая обработка запросов, поддержка асинхронного программирования и интеграция с различными библиотеками делают его оптимальным выбором для построения высоконагруженных и масштабируемых сервисов [1, 2]. Однако, чтобы использовать его потенциал максимально эффективно, требуется соблюдать архитектурные подходы [3] и следовать утвержденным лучшим практикам [4]. В том числе FastAPI обладает недостатком: отсутствием паттерна Singleton («одиночки») в DI (dependency injection – инъекция зависимостей) [5], также на текущий момент создатели и сообщество FastAPI пошли по пути того, что разработчики должны уметь писать свой вариант DI для компонент внутри системы, фреймворк Litestar же предоставляет возможность DI на каждом слое системы [6].

Зачастую начинающие разработчики отходят от принципов построения правильной архитектуры [4] (в сообществе FastAPI представлены общепризнанные практики построения такой архитектуры) и не осознают её важности, что приводит к снижению производительности и устойчивости приложений. Основной целью статьи является демонстрация ключевых принципов и практик, которые помогают создавать эффективные и устойчивые сервисы на базе FastAPI.

**Теоретическая часть**

Рассмотрим разрабатываемую архитектуру FastAPI-приложения в виде системы, которая состоит из набора подсистем (компонент). Пусть множество компонентов обозначено как:

$$C = \{C_1, C_2, ..., C_n\}.$$

Каждая компонента $C_i$ выполняет определённые функции [асинхронная обработка, ORM (Object-relational Mapping – объектно-реляционное представление), кэширование, мониторинг и т. д.] и имеет определённые показатели эффективности (пропускная способность, время отклика, надёжность, масштабируемость и т. п.). Для наглядного описания взаимосвязей между компонентами используем ориентированный граф. Определим ориентированный граф:

$$G = \langle V, E \rangle,$$

где: $V = v_1, v_2, ..., v_n$ – множество вершин, каждая вершина $v_i$ соответствует компоненте $C_i$; $E \subseteq V \times V$ – множество ребер, описывающих зависимости и каналы обмена данными (например, «компонента ORM» зависит от «компоненты кэширования»).

В общем случае если существует ориентированное ребро $(v_i, v_j) \in E$, это означает, что выходные данные (или события) компоненты $C_i$ являются входными данными (или триггерами) для компоненты $C_i$.

Каждую компоненту $C_i$ можно охарактеризовать вектором показателей эффективности [5]:

$$E = (E_1, E_2, E_3, ..., E_i),$$

где $i$ – количество учитываемых критериев. Для высоконагруженных веб-приложений целесообразно рассматривать следующие критерии (примерный набор):

$E_1$ – пропускная способность (throughput), характеризует количество обрабатываемых запросов за единицу времени;

$E_2$ – время отклика (latency), среднее или максимальное время обработки запроса;

$E_3$ – надёжность (reliability), вероятность безотказной работы компоненты в заданный период или доля корректно выполненных операций;

$E_4$ – масштабируемость (scalability), характеризует способность компоненты сохранять эффективность при увеличении нагрузки;

$E_5$ – безопасность (security), степень защищённости от несанкционированного доступа или сбоев, связанных с утечкой конфиденциальных данных.

Для получения единого «числового» показателя, позволяющего сравнивать компоненты или суммировать их вклады, можно ввести интегральную функцию $f$, которая агрегирует координаты вектора $\boldsymbol{E_i}$. Для простоты оценки системы выберем линейную агрегирующую функцию:

$$f(E) = \alpha_1 \cdot E_1 + \alpha_2 \cdot E_2 + ... + \alpha_m \cdot E_m,$$

где $\alpha_k$ – весовой коэффициент для $k$-го критерия, причём

$$\sum_{k=1}^{m} \alpha_k = 1.$$

Возьмем $\alpha_k \geq 0$, выбор значений $\alpha_k$ зависит от того, насколько важен конкретный критерий для всей системы.

Пусть S – рассматриваемая система (FastAPI-приложение). Определим суммарную (интегральную) оценку эффективности M(S) как функцию от интегральных оценок всех компонент:

$$M(S) = \Phi(f_1(E), f_2(E), ..., f_n(E)).$$

Здесь Ф – функция свертки (агрегирования), для оценки экспериментальной части выбрана именно она, потому что она дает наглядную и простую в интерпретации интегральную оценку

$$M_{mean}(S) = \frac{1}{n}\sum_{i=1}^{n} f_i(E)\,.$$

**Экспериментальная часть**

Проект построен с учётом принципов модульности, что позволяет разделить код на логические части, такие как core, users и utils. Это улучшает читаемость и упрощает навигацию по проекту. Например, функциональность, связанная с пользователями, выделена в модуль users, что позволяет добавлять или изменять её изолированно от других частей проекта.

Чистое разделение ответственности является основой архитектуры. Модуль core содержит базовую инфраструктуру приложения, включая настройки, маршруты и подключение к базе данных. Модуль users отвечает за всё, что связано с пользователями, включая API, модели, схемы, репозитории и бизнес-логику. В модуле utils размещаются вспомогательные функции, такие как логирование или утилиты. Это обеспечивает чёткую организацию и позволяет сосредоточиться на каждой компоненте отдельно.

Архитектура поддерживает масштабирование за счёт простоты интеграции новых сущностей. Например, для работы с продуктами или заказами можно создать отдельную директорию, аналогичную модулю users, с её собственными моделями, схемами и бизнес-логикой. Такая организация позволяет добавлять функциональность, не затрагивая существующий код.

Для повышения гибкости и удобства используются современные инструменты. Alembic помогает управлять миграциями базы данных, храня их в отдельной директории. Конфигурация приложения хранится в файлах .env, что упрощает управление переменными окружения и процесс деплоя. Docker обеспечивает портируемость проекта и готовность к контейнеризации благодаря наличию Dockerfile и файлов docker-compose.

Версионирование API реализовано через вложенные директории, такие как users/api/v1, где размещены маршруты и другие компоненты API. Это позволяет легко поддерживать несколько версий API. При необходимости можно создать новую версию, например v2, без конфликта с существующими компонентами. Центральная структура маршрутов в файлах, таких как /api/v1/router.py, упрощает управление конечными точками.

Архитектура следует слоистому подходу, где каждый слой выполняет чётко определённую задачу. Schemas используются для определения данных и валидации входных и выходных параметров, модели описывают ORM-структуры базы данных, репозитории содержат логику работы с базой данных, а в слое services реализуется бизнес-логика. Маршруты связывают API с бизнес-логикой, обеспечивая их интеграцию. Такое разделение упрощает изменения и тестирование, так как каждый слой может обрабатываться изолированно.

Проект реализован с соблюдением лучших практик Python. Структура соответствует стандартам PEP 8 и PEP 257, что обеспечивает чистоту и читаемость кода. Конфигурация линтеров, таких как .flake8 и .pre-commit-config.yaml, помогает поддерживать качество разработки. Использование виртуального окружения (.venv) исключает конфликты зависимостей между проектами, а стандартизированные файлы requirements.txt и pyproject.toml облегчают управление библиотеками.

Документированность проекта также играет важную роль. Файл README.md обеспечивает понятную точку входа для новых разработчиков, а примеры переменных окружения в .env.example упрощают настройку конфигурации. Эта документация способствует быстрому вовлечению новых участников команды и развертыванию проекта.

Построенная архитектура обеспечивает лёгкость разработки, масштабируемость и поддерживаемость. Она позволяет добавлять новые модули или версии API, минимизируя риски ошибок и упрощая тестирование.

Такая структура соответствует принципам SOLID и чистой архитектуры, что делает её универсальной и подходящей для реальных высоконагруженных проектов. Дополнительные детали можно найти в соответствующем репозитории на github репозитории – https://github.com/abcen7/fastapi-service-template. Ниже представлена структура разработанной архитектуры.

```
fastapi-service-template/
├── alembic/
│   ├── versions/
│   │   ├── .gitkeep
│   │   └── 0f5f83324b15_added_users_table.py
│   ├── env.py
│   ├── README
│   └── script.py.mako
├── app/
│   ├── core/
│   │   ├── database/
│   │   │   ├── __init__.py
│   │   │   ├── base.py
│   │   │   └── engine.py
│   │   ├── lib/
│   │   │   ├── __init__.py
│   │   │   ├── databases.py
│   │   │   ├── fastapi_builder.py
│   │   │   ├── logger.py
│   │   │   └── prometheus.py
│   │   ├── router/
│   │   │   ├── __init__.py
│   │   │   └── router.py
│   │   ├── __init__.py
│   │   └── settings.py
│   ├── users/
│   │   ├── api/
│   │   │   ├── v1
│   │   │   │   ├── router.py
│   │   │   │   └── __init__.py
│   │   │   ├── databases.py
│   │   │   ├── fastapi_builder.py
│   │   │   ├── logger.py
│   │   │   └── prometheus.py
│   │   ├── __init__.py
│   │   ├── models.py
│   │   ├── repositories.py
│   │   ├── schemas.py
│   │   ├── services.py
│   │   └── utils/
│   │       └── __init__.py
│   ├── __init__.py
│   └── main.py
├── venv/
├── .dockerignore
├── .env
├── .env.example
├── .flake8
├── .gitignore
├── .pre-commit-config.yaml
├── alembic.ini
├── docker-compose.dev.yml
├── docker-compose.yml
├── Dockerfile
├── makefile
├── pyproject.toml
├── README.md
└── requirements.txt
```

**Оценка созданной архитектуры**

Определим компоненты, где $C_i$ – *i*-компонента системы:

$C_1$: кодовая база и модульность. Объединяет в себе всю структуру приложения: организацию слоёв, модульное разделение (core, users, utils и т.д.) и общую архитектурную идею. Именно здесь закладывается фундамент для разработки, тестирования и масштабирования.

$C_2$: Docker и контейнеризация. Включает средства контейнеризации (Docker, docker-compose), а также инструменты CI/CD, необходимые для автоматизированного развёртывания, мониторинга и оркестрации. Эта компонента обеспечивает стабильное и воспроизводимое окружение для работы кода.

$C_3$: управление данными и миграцией. Включает в себя работу с базой данных, ORM и инструменты миграции (Alembic). Компонента отвечает за целостность, эволюцию данных и согласованность структур БД, что является критическим для функционирования высоконагруженных приложений.

$C_4$: конфигурация и документация. Охватывает файлы конфигурации (.env, .env.example), документацию (README.md), а также настройки линтеров и pre-commit. Эта компонента задаёт стандарты проекта и обеспечивает прозрачность, позволяя всем участникам быстро понимать структуру и особенности системы.

$C_5$: API и интеграция. Сосредоточен на реализации версионирования API, маршрутизации и обеспечения связи между системой и внешними сервисами/клиентскими приложениями. Он гарантирует, что интерфейсы приложения гибко адаптируются под изменяющиеся требования и легко интегрируются с другими системами.

Для каждой компоненты $C_i$ введём вектор показателей $E_i$ по 5 критериям (значения в скобках – условные оценки по шкале от 0 до 10):

$E_1$ – **Пропускная способность** (Throughput).

$E_2$ – **Время отклика** (Latency).

$E_3$ – **Надёжность** (Reliability).

$E_4$ – **Масштабируемость** (Scalability).

$E_5$ – **Безопасность** (Security).

*Замечание*. Формально для времени отклика (Latency) «меньше – лучше». Однако, чтобы не усложнять, обычно её приводят к некоторой обращённой метрике (например, «обратная латентность»).

**Связи между вершинами**

$C_1 \leftrightarrow C_2$: структура кода ($C_1$) диктует, как именно контейнеризовать и развёртывать приложение ($C_2$). В то же время инфраструктурные настройки ($C_2$) влияют на организацию кода для обеспечения совместимости с серверным окружением.

$C_1 \leftrightarrow C_3$: организация кода тесно связана с документированием и конфигурацией ($C_3$), поскольку ясная архитектура требует точного описания, а документация помогает поддерживать модульность.

$C_1 \leftrightarrow C_4$: управление данными ($C_4$) интегрируется непосредственно в кодовую базу ($C_1$) через ORM и миграции, что позволяет проводить изменения в структуре данных без нарушения общей архитектуры.

$C_1 \leftrightarrow C_5$: реализация API ($C_5$) строится на принципах, заложенных в модульной архитектуре ($C_1$). Чёткое разделение слоёв упрощает создание и версионирование API.

$C_2 \leftrightarrow C_3$: корректное развертывание ($C_2$) требует точной конфигурации и актуальной документации ($C_3$), чтобы обеспечить согласованность окружения и быстроту адаптации при изменениях.

$C_2 \leftrightarrow C_4$: инфраструктура ($C_2$) должна поддерживать миграции и работу с базой данных ($C_4$), например, через автоматизированные скрипты развертывания, которые обновляют схемы БД.

$C_2 \leftrightarrow C_5$: развертывание ($C_2$) влияет на доступность API ($C_5$), а также обеспечивает корректную работу интеграционных сервисов в продуктовой среде.

$C_3 \leftrightarrow C_4$: документация и конфигурация ($C_3$) включают описания настроек БД и миграционных процессов ($C_4$), что помогает в управлении изменениями и устранении ошибок.

$C_3 \leftrightarrow C_5$: описание API, спецификации и версии ($C_5$) документируются в ($C_3$), что упрощает работу с внешними системами и повышает прозрачность разработки.

$C_4 \leftrightarrow C_5$: API ($C_5$) зависит от структуры данных и миграций ($C_4$) для обеспечения корректной передачи и обработки информации, что требует тесной интеграции между этими компонентами.

Для каждой компоненты введем примерную оценку. Для иллюстрации взяты условные (примерные) значения, чтобы показать сам принцип агрегирования и расчёта итоговой метрики. В реальном проекте такие значения должны определяться на основе метрик из мониторинга, логов и нагрузочного тестирования:

$$C_1 : E_1 = (8,8,9,8,8).$$
$$C_2 : E_1 = (8,8,8,9,8).$$
$$C_3 : E_3 = (7,7,9,8,8).$$
$$C_4 : E_4 = (7,7,8,7,8).$$
$$C_5 : E_5 = (9,9,9,9,8).$$

Интегральная оценка каждой компоненты:

$$f(E_i) = \alpha_1 \cdot E_1 + \alpha_2 \cdot E_2 + \alpha_3 \cdot E_3 + \alpha_4 \cdot E_4 + \alpha_5 \cdot E_5,$$

где все $\alpha_k$ равны по 0,2 (поскольку все критерии мы учитываем равновесными), тогда:

$$f(E_i) = 0,2(E_1 + E_2 + E_3 + E_4 + E_5).$$

Для каждой $E_i$ вычислим сумму $\sum E_i$, а далее $f(E_i)$:

$$E_1 : \sum E_1 = 8+8+9+8+8+ = 41;$$
$$f(E_1) = 0,2 \times 41 = 82;$$
$$E_2 : \sum E_2 = 8+8+8+9+9 = 42;$$
$$f(E_2) = 0,2 \times 42 = 8,4;$$
$$E_3 : \sum E_3 = 7+7+9+8+8 = 39;$$
$$f(E_3) = 0,2 \times 39 = 7,8;$$
$$E_4 : \sum E_4 = 7+7+8+7+8 = 37;$$
$$f(E_4) = 0,2 \times 37 = 7,4;$$
$$E_5 : \sum E_5 = 9+9+9+9+8 = 44;$$
$$f(E_5) = 0,2 \times 44 = 8,8.$$

**Итоговая интегральная оценка системы:**

$$M(S) = \frac{1}{5}(f(E_1) + f(E_2) + f(E_3) + f(E_4) + f(E_5));$$

$$M(S) = \frac{(8,2 + 8,4 + 7,8 + 7,4 + 8,8)}{5} = \frac{40,6}{5} \approx 8,12;$$

$$\boxed{M(S) \approx 8,12\,(\text{по}\,10\text{-балльной шкале})}.$$

**Интерпретация результатов**

Высокая итоговая оценка ~8,12 говорит о том, что описанная структура (с использованием Docker, Alembic, раздельных модулей, документированности и версионирования) действительно соответствует лучшим практикам и обеспечивает хорошую масштабируемость, надёжность и удобство в сопровождении.

Наиболее «сильные» компоненты – это ($C_2$) Docker и ($C_5$) продуманное версионирование API, которые получили интегральные оценки 8,4 и 8,8 соответственно.

Относительно «слабое место» – это ($C_4$) – документация и конфигурация, где $f(E_4) = 7,4$. Хотя балл остаётся достаточно высоким, его можно ещё улучшить, например, внедрив более полный автоматизированный CI/CD (Continuous Integration / Continuous Delivery – одна из DevOps практик, комбинация непрерывной интеграции и непрерывного развертывания программного обеспечения) и расширив формат документации.

**Заключение**

Построение эффективной архитектуры требует соблюдения множества лучших практик, начиная с правильной структуры проекта и заканчивая использованием инструментов мониторинга и кэширования. Следуя разработанной архитектуре, разработчики могут улучшить эффективность и скорость разработки на фреймворке FastAPI, в том числе, за счет созданной математической модели можно оценить эффективность архитектуры

**Библиографический список**

1. **Соболев Н.** CPython Core Developer. Новые языки программирования, опенсорс, Python, инструменты [Электронный ресурс]. URL: https://www.youtube.com/watch?v=DS7Hhg5ebf0 (Дата обращения: 05.11.2024).

2. **Овчинников В.А.** Модели и методы дискретной оптимизации: учеб. пособие. М.: Изд-во МГТУ им. Н.Э. Баумана, 2019.

3. **Проектирование высоконагруженных систем** [Электронный ресурс]. URL: https://dynamicsun.ru/blog/proektirovanie-vysokonagruzhennyh-sistem.html (Дата обращения: 05.11.2024).

4. **FastAPI** – **advantages and disadvantages** [Электронный ресурс]. URL: https://dev.to/shariqahmed525/fastapi-advantages-and-disadvantages-197o (Дата обращения: 05.11.2024).

5. **FastAPI** – **The Good, the bad and the ugly** [Электронный ресурс]. URL: https://dev.to/fuadrafid/fastapi-the-good-the-bad-and-the-ugly-20ob (Дата обращения: 05.11.2024).

6. **FastAPI best practices** [Электронный ресурс]. URL: https://github.com/zhanymkanov/fastapi-best-practices (Дата обращения: 05.11.2024).

7. **FastAPI github issues list** [Электронный ресурс]. URL: https://github.com/fastapi/fastapi/issues/504 (Дата обращения: 05.11.2024).

UDC 004.42

# BUILDING EFFICIENT ARCHITECTURE FOR HIGHLY LOADED AND SCALABLE SERVICES ON FASTAPI: PRINCIPLES AND BEST PRACTICES

**K. A. Sermyagin,** student, RSREU, Ryazan, Russia;
orcid.org/0009-0002-6078-4655, e-mail: sermyagin.abcen7@gmail.com.
**A. N. Pylkin,** Dr. in technical sciences, professor, the department of computational and applied mathematics, RSREU, Ryazan, Russia;
orcid.org/0000-0001-9925-2870, e-mail: pylkin.a.n@rsreu.ru

*In the era of growing Internet services and high-load applications, developers are faced with the need to create systems that can handle large volumes of data and users. Key aspect for the successful de-sign of such systems is the choice of appropriate architecture and tools. FastAPI stands out among other technologies for its speed, ease of use and support for modern standards.* ***The aim of the work*** *is to improve the efficiency of development in the context of FastAPI framework by developing layered service architecture.*

***Keywords****: FastAPI, web framework, API, micro-services, design, architecture.*

**DOI:** 10.21667/1995-4565-2025-92-49-56

**References**

1. **Sobolev N.** *CPython Core Developer. New programming languages, opensource, Python, instruments* [Electronic resource]. URL: https://www.youtube.com/watch?v=DS7Hhg5ebf0 (Date of access: 05.11.2024).

2. **Ovchinnikov V.A.** *Modeli i metody diskretnoj optimizacii: ucheb. posobie* (Models and methods of the discrete optimization). Moscow: Publishing house BMSTU. 2019. (Date of access: 20.12.2024).

3. **Design of high-load systems** [Electronic resource]. URL: https://dynamicsun.ru/blog/proektirovanie-vysokonagruzhennyh-sistem.html (Date of access: 05.11.2024).

4. **FastAPI – advantages and disadvantages** [Electronic resource]. URL: https://dev.to/shariqahmed525/fastapi-advantages-and-disadvantages-197o (Date of access: 05.11.2024).

5. **FastAPI – The Good, the bad and the ugly** [Electronic resource]. URL: https://dev.to/fuadrafid/fastapi-the-good-the-bad-and-the-ugly-20ob (Date of access: 05.11.2024).

6. **FastAPI best practices** [Electronic resource]. URL: https://github.com/zhanymkanov/fastapi-best-practices (Date of access: 05.11.2024).

7. **FastAPI github issues list** [Electronic resource]. URL: https://github.com/fastapi/fastapi/issues/504 (Date of access: 05.11.2024).