

РАЗРАБОТКА СЕРВИСА АВТОРИЗАЦИИ НА FASTAPI С ИСПОЛЬЗОВАНИЕМ JWT

А.И. Литвинов¹, Г.И. Будовская¹

¹ФГБОУ ВО «Воронежский государственный лесотехнический университет
имени Г.Ф. Морозова»

Аннотация. В статье рассматривается практическая реализация сервиса авторизации на базе фреймворка FastAPI с применением JWT-токенов и асинхронного взаимодействия с базой данных. Обосновывается переход от использования Keycloak к легковесному специализированному решению в условиях ограниченных ресурсов и отсутствия необходимости в избыточном функционале. Описываются ключевые компоненты архитектуры: модели данных, процесс регистрации и аутентификации пользователей, разграничение доступа на основе ролей, а также механизмы защиты маршрутов. Предложенное решение может быть интегрировано в распределенные информационные системы в качестве централизованного сервиса аутентификации.

Ключевые слова: микросервисная архитектура, авторизация, FastAPI, JWT, асинхронное программирование, ролевая модель доступа, Python.

DEVELOPMENT OF AN AUTHORIZATION SERVICE IN FASTAPI USING JWT

A.I. Litvinov¹, G.I. Budovskaya¹

¹Voronezh State University of Forestry and Technologies named after G.F. Morozov

Abstract. This article discusses the practical implementation of an authorization service based on the FastAPI framework using JWT tokens and asynchronous database interaction. The transition from using Keycloak to a lightweight specialized solution is justified under conditions of limited resources and the absence of a need for excessive functionality. The key architectural components are described: data models, the process of user registration and authentication, role-based access control, and route protection mechanisms. The proposed solution can be integrated into distributed information systems as a centralized authentication service.

Keywords: microservice architecture, authorization, FastAPI, JWT, asynchronous programming, role-based access control, Python.

Введение

В современной разработке принято, что ИС – это не одно огромное приложение, а группа связанных сервисов. Вопросы хранения учетных записей, проверки прав доступа и управления токенами требуют централизованного подхода. Выделение сервиса авторизации в отдельный микросервис позволяет основным приложениям делегировать задачи аутентификации и не реализовывать собственную логику работы с паролями.

В разрабатываемом проекте изначально использовался Keycloak. Однако в процессе эксплуатации выяснилось, что для нужд основного приложения большинство его возможностей оставались невостребованными, при этом Keycloak предъявлял значительные требования к ресурсам и усложнял установку приложения на сторонние сервера. Это послужило причиной разработки легковесного специализированного решения.

FastAPI обеспечивает автоматическую генерацию OpenAPI-документации, встроенную валидацию данных и высокую производительность. Асинхронный режим критически важен, поскольку блокирующие запросы к базе данных могут снизить пропускную способность приложения.

Для работы с базой данных выбрана SQLAlchemy в асинхронном режиме. Такой подход позволил абстрагироваться от конкретной СУБД: на этапе разработки используется SQLite, в продакшене скорее всего будет PostgreSQL, как в остальных сервисах, причем смена базы данных не требует изменения кода.

В реализации используется bcrypt с 12 раундами соляного хеширования – это оптимальный баланс между безопасностью и производительностью. Библиотека python-jose обеспечивает поддержку JWT-спецификации.

Архитектура сервиса

Основные сущности – пользователь и роль с отношением многие-ко-многим:

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    username = Column(String, unique=True, index=True)
    email = Column(String, unique=True, index=True)
    hashed_password = Column(String)
```

```
is_active = Column(Boolean, default=True)
roles = relationship("Role", secondary=user_role, lazy="selectin")
```

Использование `lazy="selectin"` предотвращает проблему N+1 запроса при загрузке пользователей с ролями.

При старте приложения автоматически создаются стандартные роли ("admin", "user"). Учетные данные администратора задаются через переменные окружения `ADMIN_USERNAME` и `ADMIN_PASSWORD`, что обеспечивает безопасность и гибкость настройки. Система готова к работе сразу после развертывания без ручного вмешательства.

Реализация аутентификации

Процесс регистрации включает проверку уникальности имени пользователя и email, хеширование пароля и автоматическое назначение роли "user":

```
@router.post("/register")
async def register(user_data: UserCreate, db: AsyncSession = Depends(get_db)):
    # Проверка уникальности
    result = await db.execute(select(User).where(User.username == user_data.username))
    if result.scalar_one_or_none():
        raise HTTPException(status_code=400, detail="Username already registered")

    # Создание пользователя
    hashed_password = get_password_hash(user_data.password)
    new_user = User(username=user_data.username, email=user_data.email,
                    hashed_password=hashed_password, is_active=True)

    # Назначение роли "user"
    result = await db.execute(select(Role).where(Role.name == "user"))
    user_role = result.scalar_one()
    new_user.roles.append(user_role)

    db.add(new_user)
    await db.commit()
```

При успешной проверке учетных данных генерируется JWT-токен с временем жизни 30 минут. В payload включается только идентификатор пользователя:

```
@router.post("/login")
async def login(credentials: UserLogin, db: AsyncSession = Depends(get_db)):
    user = await authenticate_user(db, credentials.username, credentials.password)
```

```

if not user:
    raise HTTPException(status_code=401, detail="Invalid credentials")

access_token = create_access_token(
    data={"sub": user.username},
    expires_delta=timedelta(minutes=30)
)
return {"access_token": access_token, "token_type": "bearer"}

```

Тридцатиминутный срок действия – это компромисс между безопасностью и удобством. Для долгоживущих сессий потребуется внедрение refresh-токенов.

Функция извлекает токен из заголовка Authorization, верифицирует его и загружает пользователя из базы данных:

```

async def get_current_user(
    credentials: HTTPAuthorizationCredentials = Depends(security),
    db=Depends(get_db)
) -> User:
    payload = decode_token(credentials.credentials)
    username = payload.get("sub")

    result = await db.execute(
        select(User).where(User.username == username)
        .options(selectinload(User.roles))
    )
    user = result.scalar_one_or_none()

    if not user or not user.is_active:
        raise HTTPException(status_code=401)
    return user

```

Для разграничения доступа создана функция-фабрика, проверяющая наличие хотя бы одной из требуемых ролей:

```

def require_roles(role_names: List[str]):
    async def _check(user: User = Depends(get_current_user)) -> User:
        user_roles = {r.name for r in user.roles}
        if not user_roles.intersection(role_names):
            raise HTTPException(status_code=403, detail="Insufficient permissions")
        return user
    return _check

```

Управление пользователями

Для администраторов реализован набор эндпоинтов для CRUD-операций с учетными записями. При удалении пользователей предусмотрена защита от самоудаления:

```
@router.delete("/{user_id}")
async def delete_user(
    user_id: int,
    db=Depends(get_db),
    _current_user: User = Depends(require_roles(["admin"]))
):
    if user.id == _current_user.id:
        raise HTTPException(status_code=400, detail="Cannot delete yourself")
    await db.delete(user)
    await db.commit()
```

Обычным пользователям доступен только просмотр собственного профиля через эндпоинт `/auth/users/me`.

Заключение

Предложенное решение может служить основой для построения распределенных систем, требующих централизованного управления доступом. Дальнейшее развитие сервиса может включать внедрение refresh-токенов, интеграцию с внешними OAuth2-провайдерами и реализацию черных списков токенов. При масштабировании до уровня крупных распределенных систем управление доступом может потребовать применения методов ситуационного моделирования и теории принятия решений.

Список литературы

1. Аншина, М. Л. Архитектура приложений и данных : учебное пособие / М. Л. Аншина. – Москва: РТУ МИРЭА, 2024. – 152 с. – ISBN 978-5-7339-2218-8. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/421100> (дата обращения: 16.10.2025).
2. Годзурас, Э. Docker Compose для разработчика : руководство / Э. Годзурас ; перевод с английского А. Н. Киселева. – Москва: ДМК Пресс, 2023. – 220 с. – ISBN 978-5-93700-203-7. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/348110> (дата обращения: 16.10.2025).

3. Кочер, П. С. Микросервисы и контейнеры Docker : руководство / П. С. Кочер ; перевод с английского А. Н. Киселева. – Москва: ДМК Пресс, 2019. – 240 с. – ISBN 978-5-97060-739-8. – Текст: электронный // Лань: электронно-библиотечная система. – URL: <https://e.lanbook.com/book/123710> (дата обращения: 16.10.2025).
4. Сумин, В. И. Использование ситуационного моделирования в разработке систем принятия решений для сложных организационных систем / В. И. Сумин, А. С. Кравченко, А. В. Толкачев // Моделирование систем и процессов. – 2024. – Т. 17, № 3. – С. 71-79. – DOI 10.12737/2219-0767-2024-69-77.
5. FastAPI documentation. – URL: <https://fastapi.tiangolo.com> (дата обращения: 20.02.2026).
6. Passlib documentation. – URL: <https://passlib.readthedocs.io> (дата обращения: 20.02.2026).
7. SQLAlchemy 2.0 Documentation. – URL: <https://docs.sqlalchemy.org> (дата обращения: 20.02.2026).

References

1. Anshina, M. L. Application and Data Architecture: a textbook / M. L. Anshina. – Moscow: RTU MIREA, 2024. 152 p. ISBN 978-5-7339-2218-8. – Text: electronic // Lan: electronic library system. – URL: <https://e.lanbook.com/book/421100> (date of request: 16.10.2025).
2. Godzouras, E. Docker Compose for the developer: a guide / E. Godzouras ; translated from English by A. N. Kiselyov. – Moscow: DMK Press, 2023. – 220 p. – ISBN 978-5-93700-203-7. – Text: electronic // Lan: electronic library system. – URL: <https://e.lanbook.com/book/348110> (date of request: 16.10.2025).
3. Kocher, P. S. Microservices and Docker containers: a guide / P. S. Kocher ; translated from English by A. N. Kiselyov. – Moscow: DMK Press, 2019. 240 p. ISBN 978-5-97060-739-8. – Text: electronic // Lan: electronic library system. – URL: <https://e.lanbook.com/book/123710> (date of request: 16.10.2025).
4. Sumin, V. I. The use of situational modeling in the development of decision-making systems for complex organizational systems / V. I. Sumin, A. S. Kravchenko, A.V. Tolkachev // Modeling of systems and processes. - 2024. – Vol. 17, No. 3. – pp. 71-79. – DOI 10.12737/2219-0767-2024-69-77.
5. FastAPI documentation [Electronic resource]. – Access mode: <https://fastapi.tiangolo.com> (date of request: 02/20/2026).

6. Passlib documentation [Electronic resource]. – Access mode: <https://passlib.readthedocs.io> (date of request: 02/20/2026).

7. SQLAlchemy 2.0 Documentation [Electronic resource]. – Access mode: <https://docs.sqlalchemy.org> (date of request: 02/20/2026).