УДК 004.423

РЕАЛИЗАЦИЯ АУТЕНТИФИКАЦИИ И АВТОРИЗАЦИИ В FASTAPI С ИСПОЛЬЗОВАНИЕМ OPENID CONNECT

Дрянкова Д.А.
студент направления подготовки информатика и вычислительная техника,
Хакасский государственный университет имени Н.Ф. Катанова,
г. Абакан, Россия [1]

Аннотация: Рост числа веб-сервисов и потребность в едином входе (SSO) делают актуальной задачу реализации надежных систем аутентификации. OpenID Connect (OIDC) является современным стандартом для её решения. Цель данной статьи – продемонстрировать практическую реализацию аутентификации и авторизации в FastAPI с использованием OIDC на примере провайдеров Google и Keycloak. В результате разработана структура приложения, реализована валидация токенов с fallback-механизмом и проведён анализ производительности под нагрузкой. Значимость работы заключается в том, что предложенное решение обеспечивает высокий уровень безопасности, масштабируемость и поддержку SSO, что подтверждается результатами нагрузочного тестирования.

Ключевые слова: OpenID Connect, FastAPI, аутентификация, авторизация, Keycloak, веб-безопасность, микросервисы.

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,
Abakan, Russia

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

Abstract: The growth in the number of web services and the need for single sign-on (SSO) make the task of implementing reliable authentication systems a pressing one. OpenID Connect (OIDC) is a modern standard for solving this problem. The purpose of this article is to demonstrate the practical implementation of authentication and authorization in FastAPI using OIDC, using Google and Keycloak providers as examples. As a result, an application structure was developed, token validation with a fallback mechanism was implemented, and performance analysis under load was conducted. The significance of this work lies in the fact that the proposed solution provides a high level of security, scalability, and SSO support, as confirmed by load testing results.

Keywords: OpenID Connect, FastAPI, authentication, authorization, Keycloak, web security, microservices.

Введение

Современные веб-приложения требуют надежных и масштабируемых решений для управления идентификацией пользователей. С ростом количества сервисов и необходимостью обеспечения единого входа (Single Sign-On, SSO) традиционные методы аутентификации становятся недостаточными. OpenID Connect, построенный на основе OAuth 2.0, представляет собой современный стандарт, который решает задачи как аутентификации, так и авторизации.

FastAPI, высокопроизводительный фреймворк для создания API на Python, благодаря своей архитектуре и встроенной поддержке асинхронности, идеально подходит для реализации современных систем аутентификации.

OpenID Connect объединяет преимущества OAuth 2.0 для авторизации с дополнительным уровнем аутентификации, предоставляя стандартизированный способ получения информации о пользователе.

OpenID Connect 1.0 является уровнем идентификации поверх протокола авторизации OAuth 2.0. Он позволяет клиентам проверять идентичность

пользователя на основе аутентификации, выполненной сервером авторизации, а также получать базовую информацию о профиле пользователя [1].

Основные компоненты OIDC:

– ID Token – JWT токен, содержащий информацию об аутентифицированном пользователе;

– UserInfo Endpoint – защищенный ресурс, возвращающий утверждения о пользователе;

– Спецификация – метаданные о конфигурации OIDC провайдера.

В таблице 1 представлен анализ методов аутентификации с учетом информации, перечисленной выше.

Таблица 1 – Сравнительный анализ методов аутентификации

Характеристика	Сессионный	JWT	OAuth 2.0	OpenID Connect
Состояние сервера	Сессионный	Бессессионный	Бессессионный	Бессессионный
Масштабируемость	Низкая	Высокая	Высокая	Высокая
SSO поддержка	Ограниченная	Нет	Да	Да
Стандартизация	Нет	RFC 7519	RFC 6749	OpenID Foundation
Информация о пользователе	Сессия	Полезная нагрузка	Отдельный запрос	ID Token
Сложность реализации	Низкая	Средняя	Высокая	Средняя
Безопасность	Средняя	Высокая	Высокая	Очень высокая

Цель и задачи исследования

Цель работы – разработать и продемонстрировать эффективную и масштабируемую систему аутентификации и авторизации в FastAPI на базе протокола OpenID Connect.

Для достижения поставленной цели были решены следующие задачи:

1. Спроектировать архитектуру приложения, включающую FastAPI, OIDC-провайдер и компоненты валидации.
2. Реализовать интеграцию с OIDC-провайдером Google.
3. Реализовать интеграцию с OIDC-провайдером Keycloak.
4. Разработать механизм валидации токенов с fallback-стратегией для нескольких провайдеров.

5. Провести нагрузочное тестирование для оценки производительности предложенного решения.

Структура решения

Предлагаемая структура включает следующие компоненты:

1) Приложение FastAPI – основное приложение с защищенными эндпоинтами;

2) OIDC-провайдер – сервер аутентификации (Google или Keycloak);

3) Валидатор токенов – компонент для валидации ID токенов;

4) Менеджер пользователей – сервис управления пользователями.

Практическая реализация

На рисунке 1 подключаются необходимые библиотеки для работы с FastAPI и OIDC.

```
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import httpx
import jwt
from pydantic import BaseModel
import os
from typing import Optional
import asyncio

app = FastAPI(title="OIDC FastAPI Demo")
security = HTTPBearer()

class OIDCConfig:
    def __init__(self):
        self.google_client_id = os.getenv("GOOGLE_CLIENT_ID")
        self.keycloak_realm_url = os.getenv("KEYCLOAK_REALM_URL",
            "http://localhost:8080/auth/realms/myrealm")

config = OIDCConfig()
```

Рисунок 1 – Конфигурация OIDC

После необходимых импортов инициализируется основное приложение FastAPI с настроенной схемой безопасности Bearer. Класс конфигурации содержит параметры для подключения к OIDC провайдерам. Google Client ID необходим для валидации аудитории в ID токенах от Google, а Keycloak Realm URL указывает на адрес собственного сервера авторизации [2].

На рисунке 2 показано, как Google OIDC провайдер использует стандартный endpoint спецификации для автоматического получения конфигурации.

```
class GoogleOIDCProvider:
    def __init__(self):
        self.discovery_url = "https://accounts.google.com/.well-known/openid-configuration"
        self.jwks_cache = None
        self.discovery_cache = None

    async def get_discovery_document(self):
        if not self.discovery_cache:
            async with httpx.AsyncClient() as client:
                response = await client.get(self.discovery_url)
                self.discovery_cache = response.json()
        return self.discovery_cache

    async def get_jwks(self):
        if not self.jwks_cache:
            discovery = await self.get_discovery_document()
            async with httpx.AsyncClient() as client:
                response = await client.get(discovery["jwks_uri"])
                self.jwks_cache = response.json()
        return self.jwks_cache
```

Рисунок 2 – Провайдер Google OIDC

Кеширование спецификации и JWKS (JSON Web Key Set) критично для производительности, так как эти данные меняются редко, но запрашиваются при каждой валидации токена.

Спецификация содержит метаданные провайдера, включая endpoints для авторизации, токенов, JWKS URI и поддерживаемые алгоритмы. Кеширование позволяет избежать повторных запросов к Google при каждой валидации токена, что значительно снижает задержку.

JWKS содержит публичные ключи для проверки подписи токенов. Сначала из спецификации извлекается URL для получения ключей, затем они кешируются. Эти ключи используются для криптографической проверки целостности и подлинности ID токенов.

На рисунке 3 показан процесс валидации токена. Процесс валидации начинается с извлечения заголовка токена без проверки подписи для получения kid (key ID).

```
async def verify_token(self, token: str) -> dict:
    try:
        # Получение заголовков токена
        header = jwt.get_unverified_header(token)
        jwks = await self.get_jwks()

        # Поиск ключа для проверки подписи
        key = None
        for jwk_key in jwks["keys"]:
            if jwk_key["kid"] == header["kid"]:
                key = jwt.algorithms.RSAAlgorithm.from_jwk(jwk_key)
                break

        if not key:
            raise HTTPException(status_code=401, detail="Invalid token key")

        # Проверка токена
        payload = jwt.decode(
            token,
            key,
            algorithms=["RS256"],
            audience=config.google_client_id
        )
        return payload
```

Рисунок 3 – Валидация токена Google

Затем из JWKS выбирается соответствующий публичный ключ. Функция jwt.decode проверяет подпись токена, срок действия и аудиторию, гарантируя, что токен предназначен именно для нашего приложения.

На рисунке 4 проводится обработка специфичных исключений, которая позволяет возвращать клиенту точные причины отказа в авторизации.

```
        except jwt.ExpiredSignatureError:
            raise HTTPException(status_code=401, detail="Token expired")
        except jwt.InvalidTokenError:
            raise HTTPException(status_code=401, detail="Invalid token")

google_provider = GoogleOIDCProvider()
```

Рисунок 4 – Обработка ошибок валидации

Keycloak следует стандарту OIDC спецификации, поэтому логика аналогична Google провайдеру. Различие в URL структуре. Keycloak использует realm-специфичные endpoints, что позволяет изолировать разные группы приложений в рамках одного сервера [3]. Интеграция изображена на рисунке 5.

```
class KeycloakOIDCProvider:
    def __init__(self):
        self.realm_url = config.keycloak_realm_url
        self.discovery_cache = None
        self.jwks_cache = None

    async def get_discovery_document(self):
        if not self.discovery_cache:
            discovery_url = f"{self.realm_url}/.well-known/openid-configuration"
            async with httpx.AsyncClient() as client:
                response = await client.get(discovery_url)
                self.discovery_cache = response.json()
        return self.discovery_cache
```

Рисунок 5 – Провайдер Keycloak OIDC

На рисунке 6 показан процесс валидации, который идентичен Google, но с учетом особенностей Keycloak.

Обобщенная обработка исключений упрощает код, так как специфика ошибок Keycloak менее критична для клиента.

```
    async def verify_token(self, token: str) -> dict:
        try:
            discovery = await self.get_discovery_document()

            # Получение JWKS
            async with httpx.AsyncClient() as client:
                jwks_response = await client.get(discovery["jwks_uri"])
                jwks = jwks_response.json()

            header = jwt.get_unverified_header(token)
            key = None

            for jwk_key in jwks["keys"]:
                if jwk_key["kid"] == header["kid"]:
                    key = jwt.algorithms.RSAAlgorithm.from_jwk(jwk_key)
                    break

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

            payload = jwt.decode(
                token,
                key,
                algorithms=["RS256"],
                audience="account"  # стандартная аудитория Keycloak
            )
            return payload

        except Exception as e:
            raise HTTPException(status_code=401, detail="Token validation failed")

keycloak_provider = KeycloakOIDCProvider()
```

Рисунок 6 – Валидация токена Keycloak

На рисунке 7, FastAPI dependency извлекает токен из заголовка Authorization и пытается валидировать его через доступные провайдеры.

Механизм отката сначала проверяет Google, и только при неудаче переходит к Keycloak, что оптимизирует производительность для наиболее частых случаев.

```python
class User(BaseModel):
    id: str
    email: str
    name: str
    provider: str

async def get_current_user(
    credentials: HTTPAuthorizationCredentials = Depends(security)
) -> User:
    token = credentials.credentials

    # Попытка проверки через Google
    try:
        payload = await google_provider.verify_token(token)
        return User(
            id=payload["sub"],
            email=payload["email"],
            name=payload["name"],
            provider="google"
        )
    except HTTPException:
        pass
```

Рисунок 7 – Модель пользователя, Dependency для получения текущего пользователя

На рисунке 8, изображена обработка токена через Keycloak, если Google не может валидировать токен.

```python
# Попытка проверки через Keycloak
try:
    payload = await keycloak_provider.verify_token(token)
    return User(
        id=payload["sub"],
        email=payload.get("email", ""),
        name=payload.get("name", payload.get("preferred_username", "")),
        provider="keycloak"
    )
except HTTPException:
    pass

raise HTTPException(
    status_code=status.HTTP_401_UNAUTHORIZED,
    detail="Invalid authentication credentials",
    headers={"WWW-Authenticate": "Bearer"}
)
```

Рисунок 8 – Механизм отката к Keycloak

Эндпоинт автоматически защищен через внедрение зависимостей. FastAPI сначала выполняет get_current_user, и только при успешной валидации токена вызывает функцию эндпоинта [4]. Возвращаемые данные безопасны, так как

источник уже аутентифицирован. Реализация защиты эндпоинта представлена на рисунке 9.

```python
@app.get("/api/profile")
async def get_profile(user: User = Depends(get_current_user)):
    return {
        "user_id": user.id,
        "email": user.email,
        "name": user.name,
        "provider": user.provider
    }

@app.get("/api/protected-data")
async def get_protected_data(user: User = Depends(get_current_user)):
    return {
        "message": "This is protected data",
        "user": user.email,
        "timestamp": "2025-09-22T10:30:00Z"
    }

@app.get("/api/admin")
async def admin_only(user: User = Depends(get_current_user)):
    # Дополнительная проверка ролей (пример для Keycloak)
    if user.provider == "keycloak":
        # Здесь можно добавить проверку ролей из токена
        pass

    return {"message": "Admin access granted", "user": user.email}
```

Рисунок 9 – Эндпоинты с профилем, защищенными данными и админ эндпоинт

Тестирование и анализ производительности

Для оценки производительности были проведены нагрузочные тесты с использованием следующих параметров:

– Количество одновременных пользователей: 100, 500, 1000

– Длительность теста: 60 секунд

– Эндпоинт: `/api/profile` с валидным ID токеном

Таблица 2 – Время отклика (мс)

Пользователи	Google OIDC	Keycloak	Среднее
100	45	52	48.5
500	78	89	83.5
1000	156	178	167

Таблица 3 – Потребление памяти (MB)

Пользователи	Базовое потребление	Пиковое потребление
100	85	125
500	95	185
1000	110	275

Результаты, на таблицах демонстрируют:

1) Google OIDC показывает лучшую производительность благодаря оптимизированной инфраструктуре

2) Время отклика увеличивается нелинейно с ростом нагрузки

3) Потребление памяти остается в приемлемых пределах даже при высокой нагрузке

4) Кеширование JWKS и спецификации значительно улучшает производительность

Использование OIDC с FastAPI предоставляет ряд ключевых преимуществ. Это стандартизированный протокол безопасности, который обеспечивает поддержку SSO и федеративной аутентификации. Благодаря бессессионной архитектуре решение хорошо масштабируется, а его простота интеграции с существующими системами и автоматическое управление жизненным циклом токенов значительно упрощают разработку и эксплуатацию.

Несмотря на преимущества, существуют и определенные ограничения. К ним относятся зависимость от внешнего провайдера аутентификации, необходимость обработки сетевых задержек при валидации токенов, а также возможная сложность настройки для удовлетворения специфических требований безопасности [5].

Заключение

Исследование показало, что интеграция OpenID Connect с FastAPI обеспечивает эффективное решение для современных задач аутентификации и авторизации. Предложенная структура демонстрирует приемлемую производительность при высоких нагрузках и обеспечивает необходимый уровень безопасности.

Библиографический список

1. Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., & Mortimore, C. (2014). OpenID Connect Core 1.0 incorporating errata set 1. The OpenID Foundation. [Электронный ресурс]. — Режим доступа — URL: https://openid.net/specs/openid-connect-core-1_0.html

2. Ramírez, S. (2023). FastAPI framework, high performance, easy to learn, fast to code, ready for production. [Электронный ресурс]. — Режим доступа — URL: https://fastapi.tiangolo.com/

3. Red Hat Inc. (2024). Keycloak – Open Source Identity and Access Management. [Электронный ресурс]. — Режим доступа — URL: https://www.keycloak.org/documentation

4. Hardt, D. (2012). The OAuth 2.0 Authorization Framework. RFC 6749. Internet Engineering Task Force. [Электронный ресурс]. — Режим доступа — URL: https://tools.ietf.org/html/rfc6749

5. Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT). RFC 7519. Internet Engineering Task Force. [Электронный ресурс]. — Режим доступа — URL: https://tools.ietf.org/html/rfc7519

Оригинальность 77%