ADR-001: Python FastAPI Backend

Submitters

- Luke Doyle (D00255656)
- Hafsa Moin (D00256764)

Change Log

- approved 2026-02-10

Referenced Use Case(s)

- UVote Backend Service Requirements - Concurrent voter submission handling, JWT authentication, and async database access for the UVote election system.

Context

The UVote system requires a backend framework capable of handling concurrent voter submissions during active elections. This is architecturally significant because the choice of framework determines whether the system can meet its concurrency and latency targets, and it affects every service in the stack via the shared database and security modules.

The project initially prototyped with Flask, the framework used throughout the DkIT programme. Flask's synchronous WSGI model is a documented limitation under concurrent load, as blocking database calls cause significant response time degradation. This ruled out Flask as a production option and prompted evaluation of natively asynchronous alternatives.

The team's primary language is Python (3+ years of experience). Switching to a different language for the backend would consume 4 to 6 weeks of the 12-week Stage 1 timeline, making a Python-based solution a firm constraint.

The selected approach is FastAPI, an ASGI framework built on Starlette and Pydantic. It provides native async/await support, type-hint-based request validation, and automatic OpenAPI documentation generation. All six backend microservices will follow a consistent FastAPI application structure, sharing a common asyncpg connection pool and security module.

Proposed Design

Services and modules impacted:

All six UVote microservices (auth-service, voter-service, election-service, ballot-service, result-service, frontend-service) are implemented using FastAPI. Each service follows the same application structure: a FastAPI app instance, startup/shutdown lifecycle hooks connecting to the shared asyncpg pool, and a `/health` endpoint.

A representative service entrypoint:

```
from fastapi import FastAPI
from shared.database import Database

app = FastAPI(title="Auth Service", version="1.0.0")

@app.on_event("startup")
async def startup():
    await Database.get_pool()

@app.on_event("shutdown")
async def shutdown():
    await Database.close()

@app.get("/health")
async def health():
    return {"status": "healthy", "service": "auth"}

```

New modules:

No new modules are introduced beyond the shared `database.py` and `security.py` libraries already planned. The frontend-service uses Starlette's `Jinja2Templates` for server-side HTML rendering.

API impact:

FastAPI generates an OpenAPI specification automatically from route definitions and Pydantic models. The interactive documentation is accessible at `/docs` on each service. No manual OpenAPI maintenance is required.

Configuration:

- ASGI server: Uvicorn (`uvicorn app:app --host 0.0.0.0 --port 5001`)
- Pydantic version: v2 (pinned in `requirements.txt`)
- Python version: 3.11 (matching the `python:3.11-slim` Docker base image)

DevOps impact:

Each service is containerised using the `python:3.11-slim` base image. Deployment is managed via Kubernetes Deployment manifests. Uvicorn's startup time of approximately 2 seconds satisfies container health probe requirements.

Considerations

Flask (rejected): Flask was the most familiar option given DkIT coursework. It is unsuitable for this use case because its synchronous WSGI model blocks on database calls under concurrent load. Flask-Async exists but is not considered production-grade. Workarounds using gevent add complexity without matching the reliability of a natively asynchronous framework. Flask also requires separate libraries for JWT, input validation, and OpenAPI documentation, whereas FastAPI includes these capabilities.

Node.js with Express (rejected): Express provides native async I/O via JavaScript's event loop and would meet the performance requirements. It was rejected because all infrastructure scripts, shared libraries, and DevOps tooling are written in Python. Introducing a second language increases cognitive overhead without a proportional benefit at the scale of this project.

Go with Gin (rejected): Gin would provide the best raw performance of the options considered. It was rejected because Go is entirely new to the team, and the estimated learning investment of 4 to 6 weeks is not compatible with the Stage 1 timeline. The performance difference between Go and FastAPI is not meaningful for an election system with a target of 1,000 concurrent voters.

Java with Spring Boot (rejected): Spring Boot was rejected due to JVM startup times of 10 to 30 seconds, a baseline memory footprint of approximately 200 MB per service, and the overhead of Java boilerplate. These characteristics are poorly suited to lightweight microservices running in a resource-constrained Kind cluster.

Pydantic v2 compatibility: Some third-party libraries were slow to adopt Pydantic v2 following its 2024 release. This is mitigated by pinning the Pydantic version in `requirements.txt` for each service.

Learning curve: The team had no prior exposure to FastAPI. The estimated onboarding time was one week, based on Python familiarity and the quality of FastAPI's official documentation. This was accepted as a reasonable trade-off given the performance gains.

Decision

FastAPI was selected as the backend framework for all UVote microservices.

The decision rests on four factors. First, FastAPI is the framework used in the DkIT Microservices module, and was recommended by the module lecturer for this type of project. This provides both direct academic support and access to relevant learning material. Second, FastAPI's native async/await model eliminates the blocking behaviour associated with Flask, allowing asyncpg connections to be used without adaptation. Published benchmarks indicate FastAPI with asyncpg achieves throughput well above 1,000 req/s at low p99 latency, compared to significantly lower figures for Flask with psycopg2 under concurrent load. Third, PyJWT and bcrypt integrate with FastAPI without wrappers or adapters, satisfying the security library requirements directly. Fourth, Pydantic-based request validation reduces the attack surface by rejecting malformed input before it reaches business logic.

A review is scheduled for the end of Stage 2 (April 2026) to assess whether FastAPI meets production requirements.

The smaller extension ecosystem relative to Flask is an accepted trade-off. Starlette's ecosystem compensates for most gaps, and the built-in features (validation, OpenAPI, async) remove the need for several Flask extensions that would otherwise be required.

Other Related ADRs

- None at this time.

References

- FastAPI Documentation
- Starlette Documentation
- asyncpg Documentation
- Pydantic v2 Migration Guide
- EdgeX Foundry ADR Template