ADR-006: JWT Authentication with bcrypt Password Hashing

Submitters

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

Change Log

- approved 2026-02-11

Referenced Use Case(s)

- UVote Admin Authentication Requirements - Stateless, cross-service authentication for admin users managing elections across the UVote microservices architecture.

Context

The UVote system requires a robust authentication mechanism for admin users who create and manage elections. The solution must work reliably across six independent microservices (auth:5001, voter/admin:5002, voting:5003, results:5004, election:5005, frontend:5000) without introducing a centralised session store that becomes a single point of failure.

Early prototyping used a basic session-cookie mechanism via FastAPI's Starlette session middleware. This approach required either sticky sessions or a shared session store such as Redis to function across microservices. When the auth-service validated a session, the election-service had no way to verify that session without querying the same store. In a Kubernetes environment where pods restart regularly during development, sessions were also lost on every restart.

The system requires two distinct authentication flows: admin authentication for election management, and voter authentication via unique one-time tokens for ballot submission (ADR-005). This ADR addresses the admin authentication flow only.

Experience with JWT-based authentication during a placement at IFM provided practical familiarity with how stateless tokens can be issued and validated across services without shared state, and how access and refresh token patterns can be applied to manage session lifetime. This directly informed the approach taken here.

Proposed Design

Services and modules impacted:

The auth-service (port 5001) issues and refreshes tokens and validates admin credentials against PostgreSQL. All other services validate JWTs locally using the shared secret. Shared utilities are provided in `shared/security.py`.

JWT token creation and password hashing (`shared/security.py`):

```
import jwt
import bcrypt
from datetime import datetime, timedelta

SECRET_KEY = os.environ.get("JWT_SECRET_KEY")
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 60
REFRESH_TOKEN_EXPIRE_HOURS = 24

def create_access_token(data: dict) -> str:
    to_encode = data.copy()
    expire = datetime.utcnow() +
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire, "type": "access"})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

```

```
def verify_password(plain_password: str, hashed_password: str) ->
bool:
    return bcrypt.checkpw(
        plain_password.encode('utf-8'),
        hashed_password.encode('utf-8')
    )

def hash_password(password: str) -> str:
    return bcrypt.hashpw(
        password.encode('utf-8'),
        bcrypt.gensalt(rounds=12)
    ).decode('utf-8')

```

FastAPI dependency for route protection (`auth-service/dependencies.py`):

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

security = HTTPBearer()

async def get_current_admin(
    credentials: HTTPAuthorizationCredentials = Depends(security)
) -> dict:
    try:
        payload = jwt.decode(
            credentials.credentials, SECRET_KEY, algorithms=
[ALGORITHM]
        )
        if payload.get("type") != "access":
            raise HTTPException(status_code=401, detail="Invalid token
type")
        return payload
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token expired")
    except jwt.InvalidTokenError:
        raise HTTPException(status_code=401, detail="Invalid token")

```

Configuration:

- JWT secret: stored in Kubernetes Secret `uvote-jwt-secret` , mounted as `JWT_SECRET_KEY` environment variable
- bcrypt cost factor: 12 (hardcoded in `hash_password()`)
- Access token lifetime: 60 minutes
- Refresh token lifetime: 24 hours
- Token algorithm: HS256 (HMAC-SHA256)
- Password minimum length: 8 characters (enforced by Pydantic model)

Integration points:

- Auth-service (5001): issues and refreshes tokens, validates credentials against PostgreSQL
- Election-service (5005): validates JWT to authorise election creation and management
- Voter/admin-service (5002): validates JWT for voter registration and token distribution
- `shared/security.py` : provides `create_access_token()` , `verify_password()` , `hash_password()`
- ADR-007 (Audit): JWT claims extracted and included in audit log entries
- Kubernetes Secret: `uvote-jwt-secret` distributed to all service pods

Notes on voter authentication:

The voter authentication flow is separate from admin JWT authentication. Voters receive unique one-time access tokens generated by the admin via the voter-service. These tokens are not JWTs; they are database-backed tokens invalidated after a single use. This ADR applies exclusively to the admin authentication flow.

Considerations

Session cookies with shared Redis store (not selected): Traditional server-side session management stores a session ID in an HTTP-only cookie and maintains session data in a shared Redis instance. This approach requires all services to have network access to Redis for cross-service session validation, adds a Redis pod to the cluster, and makes Redis a single point of failure. If Redis is unavailable, no admin can authenticate to any service. This was the approach used in early prototyping and its limitations in a multi-service, frequently-restarting Kubernetes environment were the direct motivation for evaluating alternatives.

OAuth 2.0 with an external identity provider (not selected): Delegating authentication to a provider such as Auth0 or Google Identity Platform is the industry standard for production systems. It was not selected for this project because it introduces an external network dependency (an outage at the provider blocks all admin authentication), adds implementation complexity disproportionate to the number of admin users, and conflicts with the requirement for a self-contained system. Free tier limits on services such as Auth0 also risk incurring costs if exceeded.

SAML 2.0 (not selected): SAML is an enterprise SSO protocol requiring an identity provider deployment such as Keycloak. Keycloak alone requires approximately 512MB of memory and significant configuration overhead. SAML is designed for authentication across organisational boundaries and provides no meaningful benefit for a system with one to three admin users within a single Kubernetes cluster.

No immediate token revocation: If an admin JWT is compromised, it remains valid until expiry (up to one hour). This is mitigated by the short access token lifetime. A token blacklist backed by an in-memory set could be introduced in Stage 2 if a security audit identifies this as a priority.

Secret rotation: Changing the HS256 secret invalidates all outstanding tokens. Secret rotation should be coordinated during maintenance windows when no elections are active.

Symmetric signing (HS256 vs RS256): HS256 was chosen over RS256 because all services reside within the same Kubernetes cluster and trust boundary. If UVote were extended to support external service integrations such as third-party result verification, migrating to RS256 with public key distribution would be advisable.

Decision

JWT with HS256 signing and bcrypt password hashing was selected for admin authentication in UVote.

The decisive factor is stateless cross-service validation. The auth-service issues a JWT containing the admin's user ID and role. Any other service (election, voting, results) can validate this token using the shared HS256 secret without making a network call to the auth-service. This eliminates the auth-service as a runtime dependency for token validation and removes the need for a shared session store.

Experience with JWT-based authentication during a placement at IFM provided direct familiarity with this pattern in a production microservices context, giving confidence in its suitability for the UVote architecture.

bcrypt at cost factor 12 provides strong password hashing. The built-in per-password salt prevents rainbow table attacks, and the computational cost of each hash makes brute-force attacks impractical. Access tokens expire after one hour, limiting the window in which a stolen token can be used. Refresh tokens with a 24-hour expiry allow admins to maintain sessions during active election management without re-entering credentials.

The trade-off of no immediate token revocation in exchange for eliminating Redis as an infrastructure dependency is accepted. For a system with a small number of admin users, the one-hour maximum exposure window for a compromised token is reasonable. The overall deployment is simpler, the attack surface is smaller, and there is no additional infrastructure to fail.

HS256 is appropriate for the current architecture because all services operate within the same cluster trust boundary. This decision should be revisited if the system is extended to validate tokens outside that boundary.

A review is scheduled for the end of Stage 2 (April 2026) to evaluate whether token blacklisting is needed based on security audit findings.

Other Related ADRs

- ADR-001: Python FastAPI Backend - framework used for auth-service implementation
- ADR-005: Token-Based Voter Authentication - separate authentication flow for voters
- ADR-007: Audit Logging - authentication events logged here
- ADR-008: Microservices Architecture - cross-service authentication requirement

References

- RFC 7519 - JSON Web Tokens
- PyJWT Documentation
- OWASP Authentication Cheat Sheet
- bcrypt - Wikipedia
- FastAPI Security Documentation
- EdgeX Foundry ADR Template