Initialize widget with token (UNSECURE)

UI

|

| 1. Call Java endpoint with SECRET → receive JWT

v

UI

|

| 2. Call Python API with JWT in Authorization header

v

Python Service

|

| 3. Validate JWT using shared secret

| 4. Forward same JWT to Java APIs

v

Java Service

|

| 5. Validate JWT again

v

Return response

This is **not secure.**

Reasons:

- UI code can be inspected
- Browser dev tools expose requests
- Anyone can extract the secret
- Attackers could generate unlimited JWTs

This completely bypasses security.

Recommended Secure Flow (Minimal Change)

Instead of UI calling /service-token:

UI
 |
 | normal request
 v
Python Service
 |
 | request service-token from Java
 v
Java Auth Endpoint
 |
 | returns JWT
 v
Python -> Java APIs

Flow:

UI -> Python
Python -> Java /service-token (secret stored in Python)
Python -> Java API (JWT)

Benefits:

✓ secret stays backend
✓ UI cannot mint tokens
✓ service-to-service auth preserved
✓ minimal changes

Again, this flow matches the Integration User flow.

Key Differences are as follows,

Aspect	Integration User	Service Token
Authentication method	Username + Password	client_id / shared secret
Auth endpoint	/login	/service-token
User in DB	Required	Not required

Integration user:

POST /login
username=python_app
password=****

Service token:

POST /service-token
X-Service-Key: secret

Aspect	Integration User	Service Token
Token subject	actual DB user	service identity
Example subject	python_user,email,APP_A,SERVICE	python_service,internal,APP_A,SERVICE

Service token clearly represents **machine identity**.

Token Generation Logic

Aspect	Integration User	Service Token
Uses existing login flow	Yes	No
Requires new endpoint	No	Yes
Uses authentication manager	Yes	No

Integration user requires **zero code changes**.

Service token requires **custom endpoint**.

Token Management in Python

Both approaches behave almost the same once the token is issued.

Python flow:

get_token()
cache_token()
call_java_api()

So operational behavior is identical.

Specific Constraints

Because:

appId differs between apps

Python must obtain **separate tokens anyway**.

So both architectures become:

Integration user:

Python -> login App A
Python -> login App B

Service token:

Python -> service-token App A
Python -> service-token App B

So operationally, they behave **almost the same**.

PYTHON LEVEL SECURITY

1. Integration User Approach

Flow: Python logs in as an integration user → obtains JWT → calls Java APIs
Observation: Secures only Python → Java calls. Python API endpoints themselves remain unprotected. Direct calls to Python could bypass security. The JWT is only for outbound requests.
Implication: Python APIs require explicit authentication/authorization (e.g., API keys, JWTs, OAuth2, or mutual TLS).

---

2. Service Token Endpoint Approach

Flow: Python requests service token from Java → receives JWT → calls Java APIs
Observation: Similar to integration user; only secures Python → Java calls. Python endpoints remain exposed if no security is implemented.
Implication: Python APIs still need explicit security (token validation, API gateway, IP restrictions, rate limiting, etc.).

Conclusion

- Both approaches **do not address Python API security**.
- Regardless of Integration User or Service Token, you **must implement explicit security** on Python APIs.