ÉTUDE APPLIQUÉE & COURS

Concevoir, durcir et distribuer un framework backend FastAPI : `shaapi`

Un cours « Build Your Own shaapi », des fondations à la mise en production.

Shalom Tehe — Kaanari
2026

`shaapi 0.3.0 · MIT`

Résumé

Concevoir un backend web prêt pour la production — authentification, base de données migrée, autorisation par rôles, stockage, conteneurisation, et surtout une posture de sécurité saine — demande une expertise que la plupart des développeurs débutants n'ont pas encore. Résultat : on déploie des API exposées (secrets par défaut, ports de base de données ouverts sur Internet, conteneurs en root), ou bien on passe des semaines à réassembler les mêmes briques avant d'écrire la première ligne de logique métier.

Ce document présente **shaapi**, un framework et un générateur de projets pip-installable pour FastAPI — « django-admin, mais pour FastAPI » — conçu autour d'un principe directeur : être **sécurisé par défaut et impossible à déployer naïvement de façon non sécurisée**. shaapi 0.3.0 se compose de trois modules complémentaires :

- **shaapi** — le cœur : génère un projet FastAPI propre (SQLAlchemy 2 async + Alembic, Postgres, Redis, JWT, RBAC Casbin, stockage objet, Docker) durci par défaut (garde-fou de production fail-fast, conteneur non-root, anti brute-force) ;
- **shaops** — l'outillage de production : génère une configuration durcie (datastores fermés au réseau public, backups), des scripts de provisionnement VPS, et un modèle à deux branches dev/prod ;
- **shasec** — le banc de test de sécurité : audite un projet et attaque une API en fonctionnement (forge de JWT, routes non protégées, rate-limit, ports exposés) pour prouver qu'elle résiste.

Au-delà de l'outil, ce document est un cours en huit séances : la **Partie I** reconstruit shaapi de zéro (« Build Your Own shaapi ») en enseignant au passage les fondations de FastAPI et de l'ingénierie backend ; la **Partie II** montre comment s'en servir pour livrer n'importe quel projet, de la première commande jusqu'au déploiement durci et audité — en incluant la publication d'un paquet sur PyPI.

Mots-clés : FastAPI, framework backend, scaffolding, sécurité par défaut, Docker, DevSecOps, packaging Python, PyPI, pédagogie logicielle.

Table des matières

Préambule

Résumé **2**

Comment lire ce document **3**

Chapitre 0 — Introduction et positionnement **5**

- 0.1 Le problème **5**
- 0.2 État de l'art **5**
- 0.3 Contribution **5**
- 0.4 Méthodologie **5**

Partie I — Construire son propre shaapi

Séance 1 — Fondations FastAPI et l'idée d'un framework **7**

Séance 2 — Données : SQLAlchemy 2, Alembic et Postgres **9**

Séance 3 — Authentification et sécurité applicative **11**

Séance 4 — Industrialisation : Docker, Redis, stockage, observabilité **15**

Séance 5 — Du boilerplate au framework pip + publication PyPI **17**

Partie II — Utiliser shaapi sur n'importe quel projet

Séance 6 — Scaffolder et développer une fonctionnalité **21**

Séance 7 — Mise en production avec shaops **24**

Séance 8 — Sécurité offensive avec shasec + projet final **26**

Annexes et références

Conclusion et travaux futurs **29**

Annexe A — Aide-mémoire CLI **30**

Annexe B — Checklist de publication PyPI **31**

Annexe C — Catalogue des contrôles sec audit **32**

Annexe D — Dépannage **33**

Références **34**

Comment lire ce document

Vous êtes…	Lisez en priorité
Curieux du *pourquoi*	Chapitre 0 (problème, état de l'art, contribution)
Formateur / étudiant	Parties I et II (les 8 séances)
Pressé de livrer un projet	Partie II (Séances 6 à 8)

Mainteneur de paquets	Séance 5 (packaging & PyPI)

Chaque séance suit la même trame : **Objectifs** → **Théorie** → **Démonstration guidée** → **Travaux pratiques (TP)** → **Livrable** → **Pour aller plus loin**. Les blocs de code sont réels (extraits du code source de shaapi) ; les TP sont pensés pour ~3 heures.

> **CONVENTIONS**
>
> `shaapi …` désigne la commande de la CLI. Les chemins sont donnés depuis la racine d'un projet généré. Les exemples shell supposent un terminal Unix ; les notes Windows/WSL sont signalées.

Chapitre 0 — Introduction et positionnement

0.1 Le problème

FastAPI est un excellent micro-framework : rapide, typé, asynchrone. Mais il est volontairement non-opinionné — il ne dit rien sur l'organisation du code, la base de données, l'authentification, les migrations, le déploiement ni la sécurité. Pour passer d'un « hello world » à un service réel, il faut prendre des dizaines de décisions d'architecture correctes. Deux écueils en découlent :

1. **La barrière à l'entrée.** Le débutant doit maîtriser SQLAlchemy, Alembic, JWT, un système de permissions, Docker, Redis, un stockage objet… avant d'écrire sa logique métier. Beaucoup abandonnent ou copient des tutoriels incohérents entre eux.
2. **La dette de sécurité silencieuse.** Les boilerplates existants démarrent « out of the box » avec des secrets par défaut, des bases de données mappées sur l'hôte, des conteneurs en root, des cookies non sécurisés. Rien n'empêche de déployer tel quel — et c'est ce qui arrive.

> **THÈSE DE CE TRAVAIL**
>
> Un framework de scaffolding ne doit pas seulement faire gagner du temps ; il doit rendre la **voie sécurisée plus facile que la voie non sécurisée**, et refuser activement les configurations dangereuses en production.

0.2 État de l'art

Outil	Apport	Limite (vis-à-vis de notre thèse)
`django-admin startproject`	Scaffolding canonique, batteries incluses	Écosystème Django (pas FastAPI/async)
`cookiecutter` (templates)	Génération paramétrable	Inerte : aucune garantie de sécurité, pas d'outillage prod/sécu
`full-stack-fastapi-template`	Bonne base FastAPI + Docker	Pas de garde-fou prod fail-fast, pas de banc d'attaque intégré
Boilerplates communautaires	Démarrage rapide	Secrets par défaut « qui marchent », exposition réseau, root

L'originalité de shaapi n'est pas « encore un template » : c'est la **boucle fermée scaffolding → durcissement → preuve**. Le générateur produit un projet sûr par défaut ; un module (shaops) industrialise la mise en production ; un module (shasec) vérifie empiriquement que la cible résiste.

0.3 Contribution

1. Un **générateur pip-installable** (`shaapi new`) produisant un backend FastAPI complet et sécurisé par défaut.
2. Un **garde-fou de production fail-fast** : l'application refuse de démarrer hors développement tant qu'un secret par défaut subsiste — la classe de vulnérabilité « secret par défaut en prod » devient structurellement impossible.
3. **shaops** : génération d'artefacts de production durcis (overlay Docker fermant les datastores, scripts VPS, secrets) et un modèle de branches config-only dev/prod.
4. **shasec** : un testeur de sécurité dependency-free (audit statique + sondes dynamiques) servant aussi d'oracle de non-régression sécurité (utilisable en CI).
5. Une **méthode pédagogique** : reconstruire le framework enseigne, de fait, l'ingénierie backend moderne.

0.4 Méthodologie

Le document procède par **reconstruction guidée**. Plutôt que de présenter shaapi comme une boîte noire, on rebâtit chacune de ses briques, en validant empiriquement chaque propriété (par exemple : montrer qu'un JWT forgé avec le secret par défaut est rejeté, mesurer que les ports de base de données sont fermés en production). Les affirmations de sécurité sont **testées, pas postulées**.

PARTIE I

Construire son propre shaapi

> *Objectif de la partie : comprendre chaque décision d'architecture de shaapi en la réimplémentant, et en tirer les fondations réutilisables de l'ingénierie backend FastAPI.*

Séance 1 — Fondations FastAPI et l'idée d'un framework

OBJECTIFS

- Comprendre ASGI, l'asynchronisme, et le modèle de validation Pydantic.
- Maîtriser l'injection de dépendances de FastAPI (le mécanisme central).
- Saisir pourquoi une architecture en couches, et laquelle shaapi adopte.

1.1 ASGI et async, en une page

Un serveur WSGI (Flask, Django classique) traite une requête par thread/process bloquant. ASGI (le standard derrière FastAPI/Starlette) permet de gérer des milliers de connexions concurrentes sur une seule boucle d'événements : pendant qu'une requête attend la base de données, le serveur en sert d'autres. D'où la règle d'or :

> **RÈGLE D'OR**
>
> Tout I/O sur le chemin d'une requête (DB, Redis, HTTP sortant) doit être `async`. Un appel bloquant (un `time.sleep`, une lib synchrone) gèle toute la boucle.

1.2 Premier service

```python
# main.py
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI(title="Hello API")

class UserIn(BaseModel):
    email: EmailStr
    age: int

@app.post("/users")
async def create_user(user: UserIn):
    # `user` est déjà validé : email bien formé, age entier.
    return {"ok": True, "email": user.email}
```

```
uvicorn main:app --reload   # http://127.0.0.1:8000/docs
```

Deux idées majeures apparaissent déjà :

- **Pydantic** transforme une annotation de type en contrat validé à l'exécution (et en schéma OpenAPI/Swagger automatique).
- La **documentation interactive** (`/docs`) est générée gratuitement — un atout que shaapi gate en production (voir Séance 5).

1.3 L'injection de dépendances : le cœur de FastAPI

Une dépendance est une fonction dont le résultat est injecté dans la route. C'est ainsi que shaapi branche la session de base de données, l'utilisateur courant, et les contrôles d'autorisation :

```
from fastapi import Depends, HTTPException

async def get_db():
    async with async_session() as session:   # ouverture
        yield session                         # injection
    # fermeture automatique après la requête

async def get_current_user(db=Depends(get_db), token: str = Depends(oauth2)):
    user = await lookup(db, token)
    if not user:
        raise HTTPException(401, "Token invalide")
    return user

@app.get("/me")
async def me(user=Depends(get_current_user)):
    return user
```

`Depends` compose : `me` dépend de `get_current_user` qui dépend de `get_db`. FastAPI résout le graphe, gère le cycle de vie (le `yield` ferme la session), et documente les exigences (ex. l'en-tête d'autorisation) dans Swagger.

1.4 Pourquoi une architecture en couches

Un fichier `main.py` de 2000 lignes est ingérable. shaapi impose une séparation des responsabilités, héritée des bonnes pratiques *clean architecture* :

```
backend/
├── app/        # routes (la "couche transport" HTTP)
├── schemas/    # contrats d'entrée/sortie (Pydantic)
├── crud/       # accès données (requêtes SQLAlchemy réutilisables)
├── models/     # tables (SQLAlchemy ORM)
├── common/     # transverse : sécurité, réponses, exceptions, i18n…
└── core/       # configuration, démarrage de l'app
```

Le flux d'une requête : route (valide via schema) → service/crud (logique + accès models) → réponse (sérialisée via schema). Chaque couche est testable et remplaçable.

TP 1

1. Écrire une API *notes* en mémoire (liste Python) : `POST /notes`, `GET /notes`, `GET /notes/{id}` avec un schema Pydantic `NoteIn/NoteOut`.
2. Ajouter une dépendance `get_request_id()` qui génère un identifiant par requête et l'injecte dans chaque réponse.
3. Observer le schéma généré sur `/docs`.

Livrable. `notes-api` validée, documentée, avec une première dépendance maison.

Pour aller plus loin. Différence `def` vs `async def` dans une route et impact sur la boucle ; Pydantic v2 : `model_config`, validateurs (`@field_validator`).

Séance 2 — Données : SQLAlchemy 2 (async), Alembic et Postgres

OBJECTIFS

- Définir des modèles ORM et exécuter des requêtes asynchrones.
- Comprendre le pattern CRUD réutilisable de shaapi.
- Versionner le schéma avec Alembic (générer / prévisualiser / appliquer).

2.1 Modèles et session asynchrone

```python
# models/post.py
from sqlalchemy.orm import Mapped, mapped_column
from backend.common.model import Base   # déclarative de base partagée

class Post(Base):
    __tablename__ = "post"
    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(index=True)
    content: Mapped[str]
```

```python
# database/db_postgres.py (extrait simplifié)
from sqlalchemy.ext.asyncio import create_async_engine, async_sessionmaker

engine = create_async_engine(settings.postgres_dsn, echo=settings.POSTGRES_ECHO)
async_session = async_sessionmaker(engine, expire_on_commit=False)
```

Le DSN (`postgresql+asyncpg://user:pass@host/db`) utilise le driver asyncpg. `expire_on_commit=False` évite des requêtes surprises après commit.

2.2 Le pattern CRUD

Plutôt que de réécrire les mêmes SELECT/INSERT, shaapi factorise un `crud_base` générique (via `sqlalchemy-crud-plus`) dont héritent les DAO :

```
# crud/crud_post.py
from sqlalchemy_crud_plus import CRUDPlus
from backend.models.post import Post

class CRUDPost(CRUDPlus[Post]):
    ...

post_dao = CRUDPost(Post)

# usage dans une route
async def list_posts(db=Depends(get_db)):
    return await post_dao.select_models(db)
```

On obtient `create / select / update / delete` typés sans duplication ; les DAO ne contiennent que les requêtes spécifiques.

2.3 Migrations avec Alembic

Une base de données évolue : Alembic versionne ces évolutions. shaapi détecte automatiquement les nouveaux modèles (`models/__init__.py` les enregistre) pour l'autogénération.

```
shaapi db generate -m "add post table"   # autogénère une révision
shaapi db preview                         # affiche le SQL sans l'exécuter
shaapi db apply                           # alembic upgrade head
shaapi db pending                         # révision courante vs. head
```

> **DEV VS PROD**
>
> En développement, shaapi peut créer les tables au démarrage (`DB_AUTO_CREATE=true`) pour un premier essai sans friction. En production, on met `DB_AUTO_CREATE=false` et on s'appuie uniquement sur les migrations Alembic — reproductible et auditable.

2.4 Postgres en conteneur

```
# extrait docker-compose.yml
postgres:
  image: postgres:16-alpine
  environment:
    - POSTGRES_USER=${POSTGRES_USER:-postgres}
    - POSTGRES_PASSWORD=${POSTGRES_PASSWORD:-postgres}
    - POSTGRES_DB=${POSTGRES_DATABASE:-shaapi}
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER:-postgres}"]
```

Le healthcheck permet à l'API d'attendre que la base soit réellement prête (`depends_on: condition: service_healthy`) — fin des « connection refused » au démarrage.

TP 2

1. Créer le modèle `Post`, générer et appliquer la migration.
2. Écrire `crud_post` et brancher `POST /posts` + `GET /posts` sur Postgres.
3. Modifier le modèle (ajouter `published: bool`), régénérer une migration, inspecter le SQL avec `db preview` avant d'appliquer.

Livrable. API CRUD persistante sur Postgres, avec deux migrations versionnées.

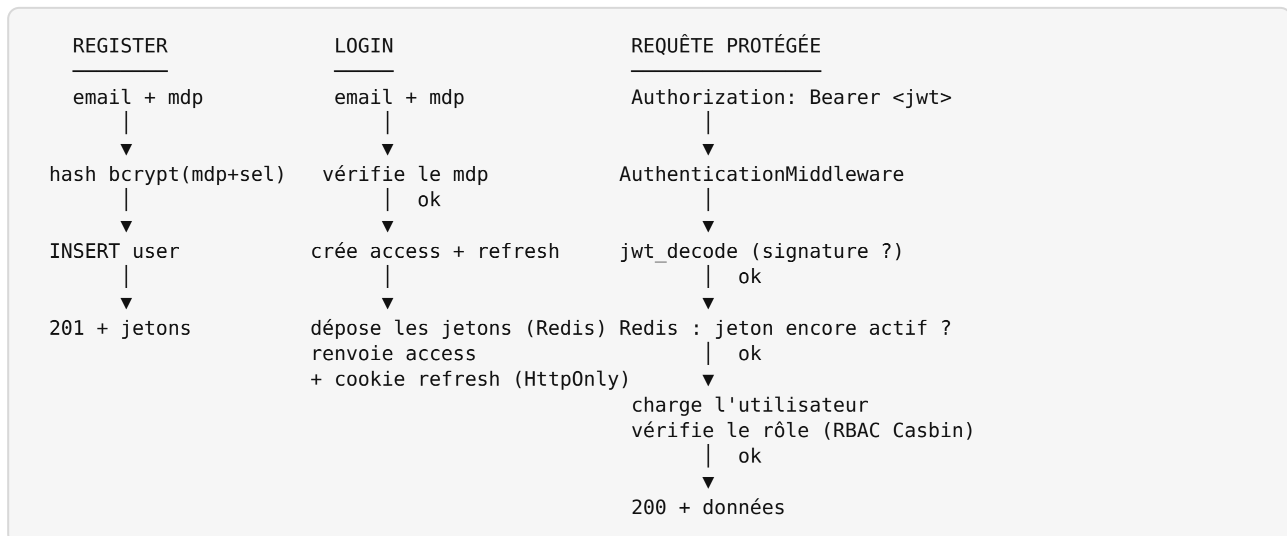
Pour aller plus loin. Relations (`relationship`, clés étrangères) et chargement (`selectinload`) ; transactions : `async with db.begin()`.

Séance 3 — Authentification et sécurité applicative

OBJECTIFS

- Stocker des mots de passe correctement (bcrypt + sel).
- Émettre et vérifier des JWT, avec révocation côté Redis.
- Mettre en place une autorisation par rôles (RBAC) avec Casbin.

LE FLUX D'AUTHENTIFICATION EN UN SCHÉMA



Trois portes successives gardent une route protégée : **signature valide → jeton encore actif (Redis) → permission suffisante (RBAC)**. Une seule qui cède et l'accès est refusé (401/403).

3.1 Mots de passe : ne jamais stocker en clair

Un mot de passe ne se stocke jamais en clair ni avec un hash rapide (MD5, SHA-1). On utilise une fonction lente et salée — **bcrypt** :

```python
# common/security/jwt.py (extrait)
from passlib.context import CryptContext
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def get_hash_password(password: str) -> str:
    return pwd_context.hash(password)

def password_verify(plain: str, hashed: str) -> bool:
    return pwd_context.verify(plain, hashed)
```

bcrypt intègre un coût (facteur de travail) qui rend le brute-force coûteux. shaapi ajoute un sel applicatif par utilisateur — défense supplémentaire.

3.2 JWT : jeton signé, mais tracé côté serveur

Un JWT est un jeton signé (HMAC-SHA256 ici) contenant un sujet (`sub`) et une expiration (`exp`). Le serveur vérifie la signature avec sa clé secrète.

```python
def create_access_token(sub: str) -> str:
    payload = {"sub": sub, "exp": now() + ACCESS_TTL}
    return jwt.encode(payload, settings.TOKEN_SECRET_KEY, "HS256")

def jwt_decode(token: str) -> str:
    payload = jwt.decode(token, settings.TOKEN_SECRET_KEY, algorithms=["HS256"])
    return payload["sub"]
```

> **DÉCISION DE CONCEPTION CRUCIALE**
>
> Un JWT « pur » est sans état : si la clé fuit, n'importe qui forge des jetons valides. shaapi ajoute une couche : chaque jeton émis est aussi stocké dans Redis. À la vérification, on exige que le jeton existe dans Redis :

```python
async def jwt_authentication(token: str) -> str:
    user_id = jwt_decode(token)                         # 1) signature valide ?
    key = f"{TOKEN_REDIS_PREFIX}:{user_id}:{token}"
    if not await redis_client.get(key):                 # 2) jeton encore actif ?
        raise TokenError("Token has expired")
    return user_id
```

Cette défense en profondeur permet la **révocation** (logout, expiration côté serveur) et — on le prouvera en Séance 8 — fait qu'un jeton forgé avec une clé par défaut est rejeté même si la signature est correcte, car il n'existe pas dans Redis.

3.3 Deux jetons : access court, refresh long

Un access token à longue durée est dangereux (s'il fuit, il vaut longtemps). La solution standard : deux jetons.

Jeton	Durée	Rôle	Transport
`access`	courte (~1 j)	accède aux routes	en-tête `Authorization: Bearer`

`refresh`	longue (~7 j)	obtient un nouvel access	cookie `HttpOnly`

Le refresh voyage dans un cookie (et non en JavaScript-accessible), ce qui le protège du vol par XSS. Sa configuration est critique :

```
response.set_cookie(
    key=settings.COOKIE_REFRESH_TOKEN_KEY,
    value=refresh_token,
    httponly=True,                                              # inaccessible au
JS (anti-XSS)
    secure=settings.COOKIE_SECURE or settings.ENVIRONMENT != "dev",   # HTTPS only en
prod
    samesite=settings.COOKIE_SAMESITE,
# 'lax' : anti-CSRF raisonnable
    max_age=settings.COOKIE_REFRESH_TOKEN_EXPIRE_SECONDS,
)
```

PIÈGE CLASSIQUE CORRIGÉ PAR SHAAPI

`SameSite=None` exige `Secure` ; sinon le cookie part en clair sur HTTP. shaapi force donc `secure=True` hors développement (et l'audit le vérifie — Séance 8).

La **rotation** : à chaque rafraîchissement, l'ancien couple est invalidé dans Redis et un nouveau couple émis. Un refresh rejoué (donc volé) échoue car il n'existe plus.

```
async def create_new_token(sub, old_token, refresh_token):
    stored = await redis_client.get(f"{REFRESH_PREFIX}:{sub}:{refresh_token}")
    if not stored or stored != refresh_token:
        raise TokenError("Refresh token expiré")          # rejoué / révoqué
    await redis_client.delete(f"{TOKEN_PREFIX}:{sub}:{old_token}")
    await redis_client.delete(f"{REFRESH_PREFIX}:{sub}:{refresh_token}")
    return await issue_new_pair(sub)                       # rotation
```

3.4 Protéger une route

```
from backend.common.security.jwt import DependsJwtAuth

@router.get("/me", dependencies=[DependsJwtAuth])
async def me(request: Request):
    return request.user
```

Un `AuthenticationMiddleware` peuple `request.user` à partir du jeton ; la dépendance `DependsJwtAuth` exige l'en-tête `Authorization: Bearer …`.

3.5 RBAC avec Casbin

L'authentification dit *qui tu es* ; l'autorisation dit *ce que tu peux faire*. shaapi utilise **Casbin**, un moteur de politiques. Deux fichiers le définissent.

Le **modèle** (`model.conf`) — la grammaire des règles :

```
[request_definition]
r = sub, obj, act        # sujet (rôle), objet (ressource), action

[policy_definition]
p = sub, obj, act

[role_definition]
g = _, _                 # g(user, role) : héritage utilisateur -> rôle

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = g(r.sub, p.sub) && keyMatch(r.obj, p.obj) && r.act == p.act
```

La **politique** (les faits, stockés en base via l'adaptateur SQLAlchemy) :

```
p, admin,  /api/v1/users/*,    GET     # le rôle admin peut lire les users
p, editor, /api/v1/articles,   POST    # editor peut créer des articles
g, alice,  admin                       # alice a le rôle admin
```

L'**évaluation** : à chaque requête, on demande à l'enforcer si (rôle, chemin, méthode) est autorisé :

```
allowed = await enforcer.enforce(user_role, request.url.path, request.method)
if not allowed:
    raise AuthorizationError("Permission refusée")
```

Avantage sur des `if user.role == "admin"` éparpillés : la politique est centralisée, modifiable à chaud (ajouter une permission = une ligne en base, sans redéploiement) et auditable. Pour les cas simples, shaapi fournit aussi un raccourci :

```
def superuser_verify(request: Request) -> bool:
    # On ne fait JAMAIS confiance au client : le rôle vient de l'utilisateur
    # rechargé depuis la base (request.user), pas d'un champ du JWT.
    if not request.user.is_superuser or not request.user.is_staff:
        raise AuthorizationError
    return True
```

RÈGLE D'OR DE L'AUTORISATION

Vérifier côté serveur, à partir de l'état rechargé en base — **jamais** à partir d'un drapeau présent dans le jeton ou le corps de la requête, qui sont sous le contrôle du client.

TP 3

1. Implémenter `register` (hash bcrypt), `login` (vérif + émission access + refresh + dépôt Redis), `me` (route protégée).
2. Ajouter une route `admin/stats` réservée au rôle admin (via `superuser_verify` ou une permission Casbin).

3. Vérifier au curl la chaîne complète :

```
B=http://localhost:8000/admin/api/v1
# (a) inscription -> renvoie un access_token
TOKEN=$(curl -s -X POST $B/auth/register \
  -H 'Content-Type: application/json' \
  -d '{"email":"e2e@test.dev","password":"Sup3rSecret1"}' | jq -r .data.ac-
cess_token)

# (b) route protégée AVEC jeton -> 200
curl -s -o /dev/null -w "me avec jeton : %{http_code}\n" \
  $B/auth/me -H "Authorization: Bearer $TOKEN"

# (c) route protégée SANS jeton -> 401/403
curl -s -o /dev/null -w "me sans jeton : %{http_code}\n" $B/auth/me

# (d) jeton bidon -> 401
curl -s -o /dev/null -w "me jeton bidon : %{http_code}\n" \
  $B/auth/me -H "Authorization: Bearer faux.jeton.invalide"
```

Résultat attendu : 200, puis 401/403, puis 401. Bonus : marteler login 6 fois et observer le passage en 429 (rate-limit, Séance 4).

Livrable. Flux d'authentification complet, jetons révocables, une route RBAC.

Pour aller plus loin. Refresh tokens et rotation ; cookies HttpOnly/Secure/SameSite ; pourquoi `is_superuser` et vérification serveur (jamais faire confiance au client).

Séance 4 — Industrialisation : Docker, Redis, stockage, observabilité

OBJECTIFS

- Empaqueter l'application en image Docker reproductible (multi-stage, uv).
- Orchestrer la stack (API, Postgres, Redis, MinIO) avec docker compose.
- Comprendre Redis (cache + rate limit), le stockage objet, les middlewares.

4.1 Dockerfile multi-stage

```
# Étape 1 : builder — résout les dépendances dans un venv isolé
FROM python:3.11-slim AS builder
COPY --from=ghcr.io/astral-sh/uv:0.11 /uv /uvx /bin/
ENV UV_PROJECT_ENVIRONMENT=/opt/venv
COPY pyproject.toml uv.lock ./
RUN uv sync --frozen --no-dev --no-install-project

# Étape 2 : runtime — image fine, sans outils de build
FROM python:3.11-slim
COPY --from=builder /opt/venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"
# Sécurité : utilisateur NON-root (voir Séance 5)
```

```
RUN groupadd --system app && useradd --system --gid app app
COPY . .
RUN chown -R app:app /app /opt/venv
USER app
ENTRYPOINT ["bash", "./backend/entrypoint-api.sh"]
```

Le multi-stage sépare la compilation (lourde) de l'exécution (légère). `uv` installe les dépendances en quelques secondes, de façon reproductible (`uv.lock`).

4.2 La stack docker-compose

Quatre services sur un réseau interne : `api` , `postgres` , `redis` , `minio` . Point clé de sécurité (approfondi en Partie II) : dans le fichier de base, les datastores **ne publient aucun port sur l'hôte** ; seules les commodités de développement (mapping de ports, bind-mount du code) vivent dans un override chargé uniquement en dev.

```
shaapi up           # build + démarre tout (dev : hot-reload)
shaapi db apply     # migrations
shaapi auth init    # premier admin
shaapi logs         # suivre les logs
```

4.3 Redis : cache et limitation de débit

Redis sert au cache, au stockage des jetons (Séance 3) et au rate limiting (anti brute-force). shaapi initialise un limiteur au démarrage et l'applique sur les routes sensibles :

```
@router.post("/login", dependencies=[Depends(RateLimiter(times=5, minutes=1))])
async def login(...): ...
```

4.4 Stockage objet (MinIO / S3)

Les fichiers (images, documents) ne vont pas en base : ils vont dans un stockage objet compatible S3 (MinIO en local, S3/GCS en prod). shaapi expose un client unifié et crée le bucket au besoin (`shaapi storage init`).

4.5 Middlewares et observabilité

shaapi empile des middlewares (ordre du bas vers le haut) : CORS, trace-id (corrélation des logs), journalisation d'accès, journal d'opérations (*opera log*), i18n. Les logs passent par `loguru` ; en conteneur, on logue sur `stdout` (lisible par `shaapi logs`) avec repli si le système de fichiers est en lecture seule.

TP 4

1. `docker compose up` → vérifier les 4 services healthy.
2. Uploader un fichier vers MinIO via une route, le retrouver dans la console MinIO (:9001).

3. Marteler login au-delà de la limite → observer le 429.

Livrable. Stack complète conteneurisée, rate-limit et stockage opérationnels.

Pour aller plus loin. Observabilité opt-in (`--monitoring`) : Prometheus/Grafana/Tempo/Loki ; le healthcheck et `depends_on: condition: service_healthy` .

Séance 5 — Du boilerplate au framework pip + publication sur PyPI

OBJECTIFS

- Transformer un projet en outil réutilisable installable par pip.
- Comprendre le scaffolding (génération + rebrand) et le durcissement.
- Publier un paquet sur PyPI (de A à Z, TestPyPI inclus).

5.1 D'un projet à un générateur

Un boilerplate qu'on copie-colle diverge à chaque projet. La bonne abstraction est un **générateur** : le code du framework vit dans un paquet `shaapi` , et un template est copié puis personnalisé à la génération.

```
# generator.py (idée)
def create_project(name, dest):
    slug = slugify(name)                          # "My API" -> "my_api"
    for src in TEMPLATE_DIR.rglob("*"):
        content = _rebrand(src.read_text(), slug)  # shaapi -> my_api
        (dest / rel).write_text(content)
```

Subtilité : certains mots `shaapi` doivent devenir le slug (titre de l'app, préfixes Redis…), mais les références à la *commande* `shaapi` doivent rester `shaapi`. shaapi résout cela par une sentinelle `{{cli}}` dans le template, restituée en `shaapi` après le rebrand — un détail qui évite des bugs subtils.

5.2 La CLI avec Typer

```
import typer
app = typer.Typer()

@app.command("new")
def new(name: str, prod: bool = typer.Option(False, "--prod")):
    """Crée un nouveau projet shaapi."""
    create_project(name, Path("."), prod=prod)
```

Déclarée dans `pyproject.toml` , elle devient un exécutable :

```
[project.scripts]
shaapi = "shaapi.cli:app"
```

5.3 Sécurité « par conception » — le garde-fou fail-fast

C'est la contribution centrale. La configuration valide ses propres invariants au démarrage : hors développement, si un secret est encore la valeur par défaut, l'application **lève une erreur et refuse de démarrer**.

```python
# core/conf.py (extrait)
from pydantic import model_validator

class Settings(BaseSettings):
    ENVIRONMENT: Literal["dev", "preprod", "prod"] = "dev"
    TOKEN_SECRET_KEY: str = "dev-insecure-change-me-token-secret-key"
    POSTGRES_PASSWORD: str = "postgres"
    MINIO_SECRET_KEY: str = "minioadmin"

    @model_validator(mode="after")
    def _enforce_production_safety(self):
        if self.ENVIRONMENT == "dev":
            return self                           # dev : zéro friction
        insecure = []
        if self.TOKEN_SECRET_KEY == "dev-insecure-change-me-token-secret-key":
            insecure.append("TOKEN_SECRET_KEY")
        if self.POSTGRES_PASSWORD == "postgres":
            insecure.append("POSTGRES_PASSWORD")
        if self.MINIO_SECRET_KEY == "minioadmin":
            insecure.append("MINIO_SECRET_KEY")
        if insecure:
            raise ValueError(f"Secrets par défaut en {self.ENVIRONMENT}: {insecure}")
        return self
```

PROPRIÉTÉ GARANTIE

« Déployer en prod avec un secret par défaut » n'est plus un oubli possible : c'est un échec au démarrage. La vulnérabilité est éliminée par construction, pas par discipline.

S'ajoutent : conteneur non-root, cookies Secure/SameSite automatiques hors dev, rate-limit sur le login, géo-IP externe désactivée par défaut, suppression de tout identifiant de démo committé.

5.4 Empaqueter : pyproject.toml

```toml
[project]
name = "shaapi"
version = "0.3.0"
requires-python = ">=3.11"
dependencies = ["typer>=0.12", "rich>=13"]   # CLE : CLI minuscule

[project.scripts]
shaapi = "shaapi.cli:app"
```

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[tool.hatch.build.targets.wheel]
packages = ["src/shaapi"]   # embarque tout le template (et prod_template)
```

Layout `src/` : le code vit dans `src/shaapi/`, ce qui évite d'importer accidentellement le paquet non installé pendant les tests. Le build backend (hatchling) ne retient que les fichiers suivis par git — d'où l'importance d'un `.gitignore` correct : un `.venv` ou un `.env` traînant ne doit jamais finir dans le wheel.

5.5 Construire le paquet (le wheel)

Un paquet Python se distribue en deux formats : la **sdist** (archive source) et le **wheel** (`.whl`, installable directement). Trois manières de construire :

```
python -m build              # standard (paquet `build`)
uv build                     # rapide (uv embarque le backend)
pip wheel . --no-deps -w dist # repli, n'a besoin que de pip
```

Toujours vérifier le contenu avant de publier (pas de secret, bons fichiers) :

```python
import zipfile
z = zipfile.ZipFile("dist/shaapi-0.3.0-py3-none-any.whl")
leaks = [n for n in z.namelist() if "/.venv/" in n or n.endswith("/.env")]
assert not leaks, leaks
```

5.6 Publier sur PyPI — pas à pas

Étape 0 — comptes et jetons. Créer un compte sur TestPyPI et PyPI. Générer un jeton API (Account settings → API tokens). On n'utilise jamais son mot de passe ; on utilise le jeton (`__token__` comme nom d'utilisateur).

Étape 1 — répéter sur TestPyPI d'abord. TestPyPI est un bac à sable : on y publie pour valider tout le processus sans polluer le vrai index.

```
pip install twine
twine upload --repository testpypi dist/*
# username:  __token__
# password: pypi-AgEN...  (le jeton TestPyPI)
```

Puis on teste l'installation depuis TestPyPI dans un venv neuf :

```
python -m venv /tmp/v && /tmp/v/bin/pip install \
  --index-url https://test.pypi.org/simple/ \
  --extra-index-url https://pypi.org/simple/ shaapi
/tmp/v/bin/shaapi --version
```

Étape 2 — publication réelle.

```
twine upload dist/*   # username __token__, password = jeton PyPI
```

> **IRRÉVERSIBLE**
>
> Une version publiée ne peut pas être réuploadée (PyPI rejette un fichier déjà existant). On *yank* au mieux une version problématique, on ne l'écrase pas. D'où : versionner proprement (SemVer : MAJEUR.MINEUR.CORRECTIF), tester sur TestPyPI, vérifier le wheel.

Étape 3 — automatiser via CI (recommandé). Plutôt que de publier depuis sa machine, on délègue à GitHub Actions, déclenché par une Release :

```yaml
# .github/workflows/publish.yml (idée)
on:
  release: { types: [published] }
jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: pipx run build
      - uses: pypa/gh-action-pypi-publish@release/v1   # OIDC, sans jeton en clair
```

Avec l'authentification OIDC (*trusted publishing*), aucun jeton n'est stocké dans le dépôt : GitHub prouve son identité à PyPI. C'est le standard moderne. Créer une Release `v0.3.0` déclenche alors la publication.

> **TP 5**
>
> 1. Construire le wheel de votre mini-framework (Séances 1–4) et vérifier son contenu (aucune fuite).
> 2. Publier sur TestPyPI, puis l'installer depuis TestPyPI dans un venv neuf et lancer sa commande.
> 3. (Bonus) Écrire un `publish.yml` déclenché par Release.

Livrable. Votre framework installable par `pip install` depuis TestPyPI.

Pour aller plus loin. Trusted publishing (OIDC) PyPI ↔ GitHub : supprimer les jetons ; classifiers, README comme long description, badges.

PARTIE II

Utiliser shaapi sur n'importe quel projet

> *Objectif de la partie : livrer un service réel — du `shaapi new` au déploiement durci et prouvé — sans réimplémenter quoi que ce soit.*

Séance 6 — Scaffolder et développer une fonctionnalité

OBJECTIFS

- Générer un projet et en comprendre la structure en quelques minutes.
- Ajouter une fonctionnalité métier de bout en bout.
- Étendre via le système de plugins.

6.1 Installer et générer

```
pip install shaapi
shaapi new "my api"   # création interactive
cd my_api
shaapi up             # build + démarre la stack (dev : hot-reload)
shaapi db apply       # migrations
shaapi auth init      # premier admin (interactif)
```

→ API sur `http://localhost:8000`, Swagger sur `http://localhost:8000/admin/api/v1/docs`.

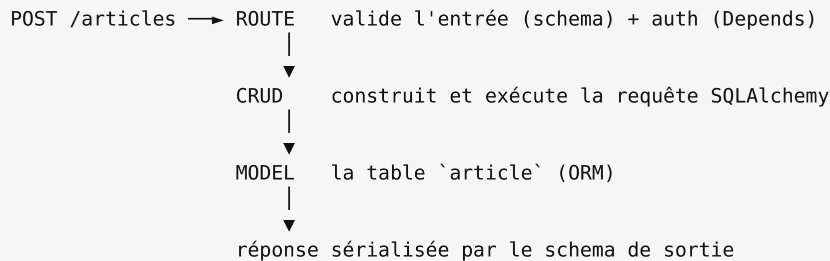
La CLI enveloppe `docker compose` directement : les mêmes commandes marchent sur Windows, macOS et Linux (pas de bash, pas de `./docker-run.sh "is not recognized"`).

6.2 Anatomie d'un projet généré

```
my_api/
├── backend/
│   ├── app/admin/api/v1/         # routes (auth, users, roles…)
│   ├── models/                   # tables ORM (déposez les vôtres ici)
│   ├── crud/  schemas/  common/  core/
│   └── seeder/                   # données de référence (aucun user par défaut)
├── docker-compose.yml            # base : datastores NON exposés
├── docker-compose.override.yml   # dev : hot-reload + ports DB locaux
├── Dockerfile                    # multi-stage, non-root
└── .env                          # config locale (gitignored)
```

6.3 Ajouter une fonctionnalité, couche par couche

On veut une ressource **Article** avec ownership : chacun crée des articles, mais ne modifie que les siens. Le flux à travers les couches :



1) Le modèle — `backend/models/article.py` (auto-détecté par Alembic) :

```
from datetime import datetime
from sqlalchemy import ForeignKey, func
from sqlalchemy.orm import Mapped, mapped_column
from backend.common.model import Base

class Article(Base):
    __tablename__ = "article"
    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(index=True)
    content: Mapped[str]
    published: Mapped[bool] = mapped_column(default=False)
    author_id: Mapped[int] = mapped_column(ForeignKey("sys_user.id"))
    created_at: Mapped[datetime] = mapped_column(server_default=func.now())
```

2) Les schémas — `backend/schemas/article.py` (le contrat d'I/O) :

```
from pydantic import BaseModel, ConfigDict

class ArticleIn(BaseModel):     # ce que le client envoie
    title: str
    content: str
    published: bool = False

class ArticleOut(BaseModel):    # ce que l'API renvoie
    model_config = ConfigDict(from_attributes=True)
    id: int
    title: str
    content: str
    published: bool
    author_id: int
```

Distinguer entrée et sortie évite de fuiter des champs internes et de laisser le client fixer ce qu'il ne devrait pas (`author_id`, `id`).

3) Le CRUD — `backend/crud/crud_article.py` :

```
from sqlalchemy import select
from sqlalchemy_crud_plus import CRUDPlus
```

```python
from backend.models.article import Article

class CRUDArticle(CRUDPlus[Article]):
    async def get_published(self, db):
        res = await db.execute(select(Article).where(Article.published.is_(True)))
        return res.scalars().all()

article_dao = CRUDArticle(Article)
```

4) La route — `backend/app/.../article.py` :

```python
from fastapi import APIRouter, Depends, Request
from backend.common.security.jwt import DependsJwtAuth
from backend.database.db_postgres import get_db
from backend.schemas.article import ArticleIn, ArticleOut

router = APIRouter(prefix="/articles", tags=["Articles"])

@router.get("", response_model=list[ArticleOut])        # public : lecture
async def list_articles(db=Depends(get_db)):
    return await article_dao.get_published(db)

@router.post("", response_model=ArticleOut, dependencies=[DependsJwtAuth])
async def create_article(obj: ArticleIn, request: Request, db=Depends(get_db)):
    # author_id vient du JWT (request.user), JAMAIS du corps de la requête
    return await article_dao.create_model(db, obj, author_id=request.user.id)

@router.delete("/{article_id}", dependencies=[DependsJwtAuth])
async def delete_article(article_id: int, request: Request, db=Depends(get_db)):
    art = await article_dao.select_model(db, article_id)
    if not art:
        raise HTTPException(404, "Introuvable")
    # OWNERSHIP : seul l'auteur (ou un admin) supprime
    if art.author_id != request.user.id and not request.user.is_superuser:
        raise HTTPException(403, "Pas votre article")
    await article_dao.delete_model(db, article_id)
    return {"deleted": article_id}
```

Puis inclure ce routeur dans l'agrégateur de l'app (`app/.../__init__.py`).

5) La migration :

```bash
shaapi db generate -m "add article table"   # Alembic détecte le nouveau modèle
shaapi db preview                            # vérifier le SQL
shaapi db apply
```

6) L'autorisation est déjà là : `DependsJwtAuth` (authentifié) + le contrôle ownership explicite dans `delete` . Pour une modération réservée aux admins, on ajouterait une dépendance de rôle (Séance 3).

> **LE RÉFLEXE À RETENIR**
>
> **model** → **schema** → **crud** → **route** → **migration**. C'est la même danse pour toute ressource. shaapi fournit les fondations ; vous n'écrivez que le métier.

6.4 Plugins

Des fonctionnalités optionnelles s'ajoutent sans copier-coller :

```
shaapi list-plugins
shaapi add advanced_auth   # ex : MFA/TOTP — copié dans backend/plugins/, deps injectées
shaapi restart api
```

TP 6

Construire une mini-API blog en suivant la danse **model → schema → crud → route → migration** :

1. Ressource `Article` (le code ci-dessus) + migration appliquée.
2. `GET /articles` public (articles publiés uniquement).
3. `POST /articles` authentifié ; `author_id` pris dans le JWT.
4. `DELETE /articles/{id}` avec ownership (auteur ou admin).
5. (Bonus) `Comment` lié à `Article` (clé étrangère + relation).

Critères de réussite :

- créer un article connecté en tant qu'utilisateur A, puis tenter de le supprimer en tant que B → 403 ;
- un visiteur anonyme lit les articles publiés mais ne peut pas en créer → 401 ;
- `shaapi db pending` ne montre aucune migration en attente.

ASTUCE DE DÉBOGAGE

Si une route protégée renvoie 403 au lieu de 200 avec un jeton valide, vérifiez que la stack tourne (`shaapi ps`) et que le jeton est bien dans Redis (`shaapi redis` → `KEYS *token*`).

Livrable. Une fonctionnalité métier complète, authentifiée et migrée.

Pour aller plus loin. Pagination (`fastapi-pagination`), réponses normalisées, gestion d'erreurs i18n.

Séance 7 — Mise en production avec shaops

OBJECTIFS

- Comprendre le modèle deux branches dev/prod (config-only).
- Générer une configuration de production durcie.
- Dérouler un déploiement VPS complet et sûr.

7.1 Le modèle dev/prod, sans dérive de code

```
shaapi new "my api" --prod
```

…crée un dépôt git à deux branches :

Branche	Contenu
`dev`	le projet de développement (vous démarrez ici)
`prod`	le même code + la configuration de production

PRINCIPE

Le code applicatif (`backend/`) est identique sur les deux branches ; seule la config diverge (overlay compose de prod, exemple d'env, scripts de déploiement). On livre en fusionnant `dev → prod` . On obtient l'isolation dev/prod sans le risque de voir le code diverger.

7.2 Durcir : shaapi ops harden

```
shaapi ops harden
```

…écrit la configuration de production :

Fichier	Rôle
`docker-compose.prod.yml`	overlay : datastores sans port hôte, `ENVIRONMENT=prod`, sidecar de backup Postgres quotidien
`.env.prod.example`	modèle d'environnement de production
`deploy/provision.sh`	installe Docker Engine + compose (Ubuntu/Debian)
`deploy/harden-os.sh`	pare-feu ufw (refuse tout sauf 22/80/443) + vérification d'exposition

Le point essentiel : on ne peut pas « retirer » un port publié via un overlay Docker (les listes de ports se concatènent). shaapi résout cela en gardant le fichier de base sécurisé (aucun port datastore) et en mettant les ports de commodité dans l'override dev. Donc `shaapi up --prod` (base + overlay prod, sans l'override dev) n'expose jamais 5432/9000.

7.3 Secrets de production

```
shaapi ops secrets --write   # génère + injecte des secrets FORTS dans .env
```

Couplé au garde-fou fail-fast (Séance 5) : un démarrage prod avec des secrets par défaut échoue ; il faut de vrais secrets, générés ici.

7.4 Déployer sur un VPS

```
# 1) Préparer le serveur (une fois)
ssh root@vps 'bash -s' < deploy/provision.sh   # installe Docker
ssh root@vps 'bash -s' < deploy/harden-os.sh   # pare-feu

# 2) Sur le serveur, depuis la branche prod
cp .env.prod.example .env
shaapi ops secrets --write
shaapi up --prod       # datastores fermés
shaapi db apply
shaapi auth init

# 3) Reverse proxy (nginx/Caddy) : TLS 443 -> 127.0.0.1:8000
```

`shaapi ops checklist` imprime à tout moment la liste de contrôle complète.

TP 7

Sur votre projet blog (Séance 6) : `--prod`, `ops harden`, `ops secrets`, puis simuler le déploiement en local (`shaapi up --prod`) et vérifier — via `docker ps` — que Postgres/MinIO ne publient aucun port hôte.

Livrable. Projet prêt pour la production, branche prod, datastores fermés, secrets forts.

Pour aller plus loin. Reverse proxy + Let's Encrypt ; sauvegardes hors-site ; rotation des secrets.

Séance 8 — Sécurité offensive avec shasec + projet final

OBJECTIFS

- Auditer un projet (statique) et attaquer une API en marche (dynamique).
- Intégrer shasec en CI comme garde-fou de non-régression.
- Boucler un projet de fin de cours : construire → durcir → attaquer → corriger.

8.1 Audit statique — shaapi sec audit

shasec code en dur la revue de sécurité du code généré. Sur un projet neuf, il ne signale que les secrets par défaut de dev (sains en dev, bloqués en prod par le garde-fou). Après `shaapi ops secrets --write`, c'est propre.

```
[CRITICAL] Default secret in use: TOKEN_SECRET_KEY
[CRITICAL] Default secret in use: POSTGRES_PASSWORD
[PASS    ] Production fail-fast guard present
```

Il signale aussi : `.env` suivi par git, identifiants committés dans les seeds, ports datastores exposés, conteneur root, cookies faibles, docs non protégées, géo-IP externe.

8.2 Attaques dynamiques

```
shaapi sec auth  http://localhost:8000/admin/api/v1
shaapi sec scan  http://localhost:8000/admin/api/v1
shaapi sec ports localhost
```

- **sec auth** — forge un JWT avec la clé par défaut publique et appelle `/auth/me`. Une API sans état qui ne vérifie que la signature renverrait 200 ; shaapi renvoie 401 car les jetons sont aussi tracés dans Redis (Séance 3). Teste aussi les routes non protégées et le rate-limit du login (doit finir en 429).
- **sec scan** — en-têtes de sécurité manquants (HSTS, X-Frame-Options…) et surface OpenAPI exposée.
- **sec ports** — accessibilité TCP des ports datastores : ouverts en dev, fermés en `--prod`.

8.3 La preuve par l'expérience

Le résultat empirique attendu (et reproductible) :

```
sec auth :
  [PASS] Forged default-secret token rejected (GET /auth/me -> 401)
  [PASS] Login is rate-limited (… 401,401,401,401,429)

sec ports (dev) :
  [HIGH] Postgres port 5432 is OPEN

sec ports (--prod) :
  [PASS] Postgres port 5432 is closed
```

> **LA BOUCLE FERMÉE**
>
> C'est la boucle fermée de notre thèse : shaapi est conçu pour résister à shasec, et shasec prouve qu'il résiste. La sécurité n'est plus une promesse, c'est un test reproductible.

8.4 En intégration continue

`sec audit` sort en code non-zéro dès qu'un résultat est HIGH/CRITICAL :

```
shaapi sec audit || exit 1   # casse le build si régression de sécurité
```

8.5 Projet final

Énoncé. Construire une API complète (blog multi-utilisateurs ou gestionnaire de tâches) :

1. Modèles + migrations + CRUD + routes protégées (Séances 2–3, 6).
2. Conteneurisée et démarrable (`shaapi up`).

3. Passée en production (`--prod`, `ops harden`, `ops secrets`) — Séance 7.
4. Auditée et attaquée avec shasec jusqu'au vert (Séance 8).
5. (Bonus) Votre propre framework dérivé publié sur TestPyPI (Séance 5).

Critères d'évaluation. Fonctionnalité correcte (30 %), qualité de l'architecture (20 %), `sec audit` sans HIGH/CRITICAL après durcissement (30 %), déploiement reproductible (20 %).

Livrable. Un projet construit, durci, attaqué et défendu — votre preuve de maîtrise.

Conclusion et travaux futurs

shaapi défend une idée simple mais structurante : **un framework de scaffolding doit rendre la voie sécurisée plus facile que la voie non sécurisée, et refuser activement les configurations dangereuses en production.** La concrétisation est une boucle fermée en trois temps — générer un projet sûr par défaut (shaapi), industrialiser un déploiement durci (shaops), puis prouver la résistance par l'attaque (shasec).

Ce que cette approche apporte, mesurable :

- la classe de vulnérabilité « secret par défaut en production » est **éliminée par construction** (garde-fou fail-fast), pas confiée à la vigilance ;
- l'exposition réseau des datastores en production est **nulle par défaut**, et cette propriété est vérifiable (`shaapi sec ports`) ;
- la posture de sécurité devient un **test reproductible** intégrable en CI, donc une propriété non-régressable.

Travaux futurs.

- **shasec 0.4.0 —** plugins optionnels (nuclei, OWASP ZAP, sqlmap) pour des campagnes d'intrusion plus profondes, en complément du cœur natif.
- En-têtes de sécurité par défaut (HSTS, CSP, X-Frame-Options) via un middleware ou un reverse proxy fourni.
- Trusted publishing (OIDC) généralisé et signatures d'artefacts.
- Étude utilisateur : mesurer la réduction du temps « du zéro au déploiement sécurisé » sur une cohorte d'étudiants — la validation empirique de la thèse pédagogique.

> **EN UN MOT**
>
> shaapi n'est pas « un template de plus » : c'est une proposition méthodologique — **secure-by-default, prouvé par l'attaque** — applicable au-delà de FastAPI.

Annexe A — Aide-mémoire CLI

SCAFFOLDING

```
shaapi new "my api"                créer un projet (interactif)
shaapi new "my api" -y             accepter les défauts
shaapi new "my api" --prod         + branches git dev/prod (config prod)
shaapi new "my api" --monitoring   + stack Prometheus/Grafana/Tempo/Loki
```

CYCLE DE VIE (ENVELOPPE DOCKER COMPOSE, MULTIPLATEFORME)

```
shaapi up [--prod] [--monitoring]   build + démarrer
shaapi down                         arrêter + supprimer les conteneurs
shaapi logs [service]               suivre les logs
shaapi restart [service]            redémarrer
shaapi ps                           état des conteneurs
shaapi shell                        bash dans le conteneur api
shaapi redis                        redis-cli
```

BASE DE DONNÉES (ALEMBIC)

```
shaapi db generate -m "message"   autogénérer une migration
shaapi db apply                   appliquer (upgrade head)
shaapi db preview                 afficher le SQL sans l'exécuter
shaapi db pending                 révision courante vs head
shaapi db shell                   psql dans le conteneur Postgres
```

AUTH / STOCKAGE

```
shaapi auth init [-e … -w …]   créer le premier admin
shaapi storage init            créer le bucket objet
```

PRODUCTION (SHAOPS)

```
shaapi ops harden            générer compose.prod.yml + .env.prod + deploy/
shaapi ops secrets [--write] générer des secrets forts (injecter dans .env)
shaapi ops checklist         checklist de mise en production
```

SÉCURITÉ (SHASEC) — CODE RETOUR ≠ 0 SI HIGH/CRITICAL

```
shaapi sec audit          audit statique du projet
shaapi sec auth  <url>    forge JWT / routes / rate-limit
shaapi sec scan  <url>    en-têtes de sécurité + surface OpenAPI
shaapi sec ports <host>   ports datastores joignables
```

PLUGINS / META

```
shaapi list-plugins | add <n> | remove <n>
shaapi --version
```

Annexe B — Checklist de publication PyPI

```
[ ] Version bumpée (SemVer) dans pyproject.toml
[ ] CHANGELOG / notes de version à jour
[ ] Working tree propre, tests verts
[ ] Build : python -m build (ou uv build / pip wheel . --no-deps -w dist)
[ ] Vérifier le contenu du wheel (AUCUN .venv, .env, secret)
[ ] twine check dist/* (métadonnées valides)
[ ] Publier sur TestPyPI : twine upload --repository testpypi dist/*
[ ] Installer depuis TestPyPI dans un venv NEUF et lancer la commande
[ ] Publier en réel : twine upload dist/* (username __token__)
    … ou créer une Release GitHub -> workflow publish.yml (OIDC)
[ ] Vérifier la page du projet sur pypi.org (README rendu, version)
[ ] Tag git annoté : git tag -a vX.Y.Z -m "…" && git push --tags
```

RAPPEL

Une version PyPI est immuable — on ne réécrit pas, on *yank* au pire. Toujours répéter sur TestPyPI d'abord.

Annexe C — Catalogue des contrôles sec audit

Contrôle	Sévérité	Remédiation
Secret encore par défaut (`TOKEN_SECRET_KEY`, `POSTGRES_PASSWORD`, `MINIO_SECRET_KEY`, `OPERA_LOG_…`)	**CRITICAL**	`shaapi ops secrets --write`
`.env` suivi par git	**CRITICAL**	`git rm --cached .env` + rotation
Identifiants committés dans `seeder/json/*`	**HIGH**	supprimer ; `shaapi auth init`
Le compose de base publie un port datastore	**HIGH**	déplacer le mapping vers l'override dev
CORS `*` avec credentials	**HIGH**	restreindre `CORS_ALLOWED_ORIGINS`
Conteneur en root	**MEDIUM**	ajouter un `USER` non-root
Cookie `SameSite=None` sans `Secure`	**MEDIUM**	`secure=True, samesite='lax'`
Docs non protégées par l'environnement	**MEDIUM**	`docs_url=None` si `ENVIRONMENT==prod`
Géo-IP externe activée	**LOW**	`IP_LOCATION_PARSE=false`
Garde-fou de production présent	***PASS***	—

Annexe D — Dépannage

Windows / PowerShell. Utiliser `Activate.ps1` (pas `activate.bat`) pour le venv ; la CLI shaapi (qui enveloppe docker compose) évite les soucis de scripts shell. Docker Desktop doit tourner.

WSL. Accès aux fichiers Windows via `/mnt/c/…`. Pratique pour tester les scripts `deploy/*.sh` en vrai Ubuntu (`bash -n script.sh` valide la syntaxe sans exécuter).

No such command 'new'. Vous avez une version trop ancienne : `pip install --upgrade shaapi` (la commande s'appelait `create-project` avant 0.2.2).

Un port semble « ouvert » malgré --prod. `shaapi sec ports` fait une vraie connexion TCP : il peut détecter un service hôte (ex. un Postgres natif) sans rapport avec la stack. Vérifier avec `docker ps` que le conteneur ne publie pas le port (`5432/tcp` = interne ; `0.0.0.0:5432->` = publié).

La stack ne démarre pas en prod. C'est probablement le garde-fou : des secrets par défaut subsistent. Lancer `shaapi ops secrets --write` et renseigner `.env`.

Références

1. FastAPI — documentation officielle. fastapi.tiangolo.com
2. Starlette (ASGI). starlette.io
3. Pydantic v2. docs.pydantic.dev
4. SQLAlchemy 2.0 (async). docs.sqlalchemy.org
5. Alembic. alembic.sqlalchemy.org
6. Casbin (RBAC). casbin.org
7. uv (gestion de paquets). github.com/astral-sh/uv
8. Packaging Python (PyPA). packaging.python.org
9. Trusted Publishing (PyPI/OIDC). docs.pypi.org/trusted-publishers
10. OWASP — Application Security Verification Standard (ASVS).
11. ip2region (base géo-IP hors-ligne). github.com/lionsoul2014/ip2region
12. Code source de shaapi. github.com/Shalom-302/shaapi

Document rédigé pour la communauté. shaapi est distribué sous licence MIT. Retours et contributions bienvenus sur le dépôt.