1. Lisibilité et clarté

Le code est globalement clair et lisible. Chaque fonction est bien nommée et les commentaires présents facilitent la compréhension. Cependant, l'ajout de docstrings complètes (notamment pour les fonctions de manipulation de la base de données) améliorerait encore la lisibilité et la maintenance.

Il serait aussi utile d'ajouter des blocs try/except pour capturer les erreurs inattendues, ce qui améliorerait la clarté et la robustesse du code.

2. Qualité du code

a) Architecture

L'architecture initiale repose sur un unique fichier app.py, ce qui nuit à la lisibilité et à la modularité.

J'aurais modifié cela pour aboutir à une séparation des responsabilités avec la création de fichiers dédiés :

- app.py pour la configuration et le lancement de l'app
- routes.py pour les endpoints
- database.py pour la gestion de la base de données
- utils.py pour les fonctions utilitaires

Cela respecte les bonnes pratiques de développement et facilite la scalabilité future.

Autre amélioration que j'aurais apportée :

- L'utilisation du Blueprint pour organiser les routes, afin de permettre une extension modulaire plus claire.
- L'utilisation @app.post() au lieu de @app.route() car qu'une seule methode n'est utilisé dans chacun des endpoints.

Enfin, dans la version initiale, la fonction init_db() est appelée directement dans le fichier principal, ce qui la fait s'exécuter même lors d'un simple import — une mauvaise pratique. J'aurais déplacé cette logique dans un @app.before_first_request, respectant le cycle de vie de l'application Flask. (Cette methode est deprecated dans les version >= 2.1.3 de Flask, mais nous sommes en 2.1.2 donc cela fonctionne avec notre version de Flask)

b) Séparation des responsabilités

Les responsabilités devraient être clairement séparées. Tout concentrer dans un seul fichier nuit à la maintenabilité. J'aurais déplacé chaque couche (logique applicative, persistence, utilitaires) dans son propre module pour faciliter la maintenance, les tests, et les évolutions futures (par exemple ajout de tests unitaires).

c) Modularité

Le code de base n'est pas modulaire. J'aurais apporté les changements suivants :

- Externaliser les routes dans un Blueprint
- Utiliser un système de gestion des connexions à la base via Flask g
- Créer des modules dédiés (voir ci-dessus)

Je recommande également de :

- Placer les Blueprints dans un sous-dossier dédié (blueprints/) si plusieurs Blueprints sont introduits. (voir doc officielle des Blueprint Flask)
- Ajouter un répertoire tests/ pour les tests unitaires.

3. Performance et optimisation

Dans le code initial, chaque endpoint ouvre et ferme manuellement une connexion SQLite, ce qui est inefficace.

J'aurais modifié cela pour adopter l'usage du contexte Flask (g) pour stocker la connexion à la base. Cela suivrait la documentation officielle et permettrait une gestion plus performante des connexions.

J'aurais également ajouté un handler teardown_appcontext pour assurer la fermeture propre et automatique des connexions à la fin de chaque requête.

Le chemin de la base est ecrit en brut lors de l'appel de la creation de la base, il serait mieux de le stocker dans app.config. Cepenant le chemin serait toujours ecrit en brut, il serait encore mieux de le lire depuis une variable d'environnement.

4. Sécurité

L'API ne valide pas la présence des champs url et short_url dans les payloads JSON, ce qui cause un crash de l'application si l'un de ces champs n'est pas fourni dans le body de la requete.

De plus, la troncature SHA256 à 6 caractères est insuffisante à terme (risque de collisions). J'aurais :

- Allongé le hash à 8 ou 10 caractères
- Ou ajouté un mécanisme de fallback en cas de doublon (par boucle ou INSERT OR IGNORE + vérification).

De plus, pour renforcer la sécurité il serait pertinent d'introduire un mécanisme de rate limiting afin de protéger contre les attaques par brute force ou ddos.

5. Testabilité et maintenance

Le code initial manque de presence de tests. J'aurais restructuré l'application comme décrit plus haut pour :

- Ajouter des tests unitaires sur utils.py
- Tester les endpoints avec notre propre client

Je recommande aussi d'ajouter un dossier tests/ avec pytest et j'aurais également généralisé l'utilisation de docstrings dans toutes les fonctions publiques.

6. Bonnes pratiques & production

Le mode debug est activé dans le commit de base (app.run(debug=True)) – Il faudrait le désactivé (debug=False) car dans le Dockerfile il est precisé que nous sommes en production et non pas developpement.

Le Dockerfile est simple et efficace, mais ce qui peut être amélioré :

- Ajouter un .dockerignore
- Utiliser ENTRYPOINT au lieu de CMD
- Lire la configuration depuis des variables d'environnement (via .env par exemple)
- Passer sur un format de docker compose pour faciliter l'ajout de nouvelles fonctionnalités si nécessaire

7. Améliorations possibles :

- Changer la méthode HTTP de /decode en GET, avec un query parameter short_url, pour se conformer à REST. Cela permettrait une séparation plus claire entre création (POST) et consultation (GET), respectant les conventions RESTful.
- Mettre en place des logs structuré pour tracer les appels, erreurs, et tentatives malveillantes.
- Gérer les collisions d'ID de manière explicite.