Documentație: Utilizarea Proxy Pattern în Aplicația Flask

Această documentație descrie utilizarea Proxy Pattern într-o aplicație Flask, explicând unde, de ce și cum a fost implementat.

Ce este Proxy Pattern?

Proxy Pattern este un design pattern structural care oferă un substitut sau un intermediar pentru un obiect real, controlând accesul la acesta. Este utilizat pentru a adăuga un nivel de abstracție în fața obiectelor complexe sau consumatoare de resurse, facilitând accesul și gestionarea lor.

Unde este folosit Proxy Pattern în Aplicație

În aplicația Flask prezentată, Proxy Pattern este utilizat pentru a abstractiza logica de acces și manipulare a datelor în următoarele module:

1. **UserProxy**: Pentru gestionarea utilizatorilor.
2. **GroupProxy**: Pentru gestionarea grupurilor.
3. **FacilityProxy**: Pentru gestionarea facilităților.
4. **ReviewProxy**: Pentru gestionarea recenziilor.
5. **SportProxy**: Pentru gestionarea sporturilor.

De ce s-a folosit Proxy Pattern?

Proxy Pattern a fost utilizat pentru a rezolva mai multe cerințe funcționale și probleme legate de performanță și scalabilitate:

1. **Optimizarea Performanței**:
    - S-a folosit caching (în combinație cu Flask-Caching) pentru a evita interogările frecvente în baza de date.
    - Proxy-ul îmbunătățește viteza aplicației prin stocarea temporară a datelor în memorie (cache).
2. **Separarea Logicii**:
    - Proxy-ul permite separarea logicii de manipulare a datelor de logica aplicației, menținând codul mai organizat și mai ușor de întreținut.
3. **Controlul Accesului**:
    - Proxy-ul gestionează accesul la obiectele din baza de date, permițând implementarea unor mecanisme suplimentare de control al accesului (e.g., caching condiționat).
4. **Reducerea Repetării Codului**:
    - Proxy-ul centralizează funcții comune, cum ar fi recuperarea, adăugarea sau ștergerea obiectelor din baza de date, reducând codul duplicat.

Cum a fost implementat Proxy Pattern?

Structura Proxy

Fiecare proxy este o clasă distinctă care acționează ca un intermediar între aplicație și baza de date. Structura generală este următoarea:

1. **Constructor**:
    - Primește o sesiune SQLAlchemy („db.session”) și un obiect de caching.
2. **Metode**:
    - Implementă funcții pentru operații CRUD (Create, Read, Update, Delete) specifice entității gestionate.
    - Utilizează caching pentru a optimiza interogările frecvente.

Exemplu:

```
class UserProxy:
  def __init__(self, db_session, cache):
    self.db_session = db_session
    self.cache = cache

  def get_user_by_id(self, user_id, refresh_cache=False):
    cache_key = f"user_{user_id}"
    if refresh_cache:
      self.cache.delete(cache_key)

    user = self.cache.get(cache_key)
    if not user:
      user = User.query.get(user_id)
      self.cache.set(cache_key, user)
    return user
```

Exemple specifice de implementare

1. UserProxy

- Gestionează accesul la utilizatori.
- Funcții disponibile:
    - get_user_by_id(user_id, refresh_cache=False)
    - is_admin(user_id)
    - delete_user(user_to_ban)
    - update_user_levels(user_id)

- add_user(new_user)
- get_user_by_email(email)
- get_users_ordered_by_behavior()

2. GroupProxy

- Gestionează grupurile și participarea utilizatorilor la acestea.
- Funcții disponibile:
  - get_group_by_id(group_id)
  - add_group(group_data)
  - delete_group(group_id)
  - add_participant(user_id, group_id)
  - remove_participant(participant)
  - get_filtered_groups(sport_filter=None, date_filter=None, facility_filter=None)
  - check_participant_exists(user_id, group_id)
  - get_group_participants(group_id)
  - get_groups_by_participant(user_id)
  - get_active_groups_by_participant(user_id)
  - get_past_groups(user_id)
  - get_participant(user_id, group_id)
  - delete_participants_by_group(group_id)
  - get_all_groups()
  - delete_group_participants(group)
  - get_groups_by_admin(admin_id)
  - get_participant_records_by_user(user_id)
  - get_groups_by_facility(facility_id)

3. FacilityProxy

- Gestionează facilitățile sportive.
- Funcții disponibile:
  - get_facility_by_id(facility_id)
  - add_facility(facility)
  - delete_facility(facility)
  - get_facilities_dict()

- get_all_facilities()
- get_facilities_by_sport(sport)

4. ReviewProxy

- Gestionează recenziile utilizatorilor.
- Funcții disponibile:
  - get_review_by_id(review_id)
  - add_review(review)
  - delete_review(review)
  - get_reviews_by_reviewed_id(reviewed_id)
  - delete_reviews_by_user(user_id)

5. SportProxy

- Gestionează sporturile disponibile.
- Funcții disponibile:
  - get_sport_by_name(sport_name)
  - add_sport(new_sport)
  - get_all_sports()
  - get_all_sports_names()

Beneficiile utilizării Proxy Pattern

1. **Performanță crescută**:
   - Cache-ul reduce numărul de interogări la baza de date.
2. **Organizare**:
   - Codul este modular, și logica este clar separata.
3. **Extensibilitate**:
   - Proxy-urile pot fi extinse ușor pentru a adăuga noi funcționalități (e.g., alte metode de filtrare).
4. **Reutilizare**:
   - Metodele din proxy-uri pot fi utilizate în diferite părți ale aplicației, reducând codul duplicat.

Considerații pentru viitor

1. **Gestionarea cache-ului**:
   - Să se implementeze politici de expirare a datelor din cache pentru a evita informații stale.

2. **Testare**:
    - Să se creeze teste unitare pentru metodele proxy pentru a asigura integritatea logicii.
3. **Extinderea funcționalității**:
    - Să se implementeze metode mai avansate de filtrare și interogare.

Această implementare a Proxy Pattern reprezintă o soluție eficientă pentru gestionarea logicii complexe și optimizarea performanței aplicației.