



**UNIVERSITATEA
TEHNICĂ
DIN CLUJ-NAPOCA**

TasteBuds

Ingineria Sistemelor

Autori: Robert-Marius Cocoroadă și Ștefan-Alexandru Onofrei
Grupa: 30237

FACULTATEA DE AUTOMATICĂ
ȘI CALCULATOARE

22 Ianuarie 2026

Cuprins

| | | |
|----------|--|-----------|
| 1 | Scurtă descriere a proiectului | 2 |
| 2 | Limbaje și tehnologii folosite | 2 |
| 3 | Cerințe funcționale | 2 |
| 3.1 | Diagrama Use-Case | 3 |
| 4 | Cerințe non-funcționale | 3 |
| 5 | Design Patterns | 4 |
| 5.1 | Factory Pattern | 4 |
| 5.1.1 | Diagrame Factory Pattern | 4 |
| 5.2 | Facade Pattern | 4 |
| 5.2.1 | Diagrame Facade Pattern | 10 |
| 6 | Logică și Algoritmi Implementați | 10 |
| 6.1 | 1. Geolocație și Integrare API | 10 |
| 6.1.1 | Descriere | 10 |
| 6.1.2 | Cod (Python Backend) | 10 |
| 6.2 | 2. Algoritmul de Matching | 10 |
| 6.2.1 | Descriere | 10 |
| 6.2.2 | Pseudo-Cod | 10 |
| 6.3 | 3. Selecția Aleatorie a Câștigătorului | 10 |
| 6.3.1 | Descriere | 10 |
| 6.3.2 | Implementare | 11 |
| 7 | Instrucțiuni de Instalare (Read Me) | 11 |

1 Scurtă descriere a proiectului

TasteBuds este o aplicație web interactivă de tip "Tinder for Food", concepută pentru a rezolva eterna problemă a grupurilor: "Unde mâncăm?". Aplicația permite utilizatorilor să creeze camere virtuale (Rooms), să invite prieteni și să voteze (Swipe Left/Right) restaurantele din apropierea lor.

Atunci când toți membrii unui grup dau "Like" (Swipe Right) aceluiași restaurant, se realizează un "Match". La finalul sesiunii, aplicația alege aleatoriu un câștigător dintre restaurantele comune și oferă indicații de orientare (Google Maps) către acesta.

Funcționalitățile cheie includ autentificarea securizată prin Google, sincronizarea în timp real a voturilor între utilizatori și salvarea istoricului camerelor vizitate.

2 Limbaje și tehnologii folosite

Arhitectura aplicației este una modernă, bazată pe microservicii și servicii cloud:

- **Frontend:** React.js (JavaScript) - pentru interfața utilizator, gestionarea stării și logica de swipe.
- **Stilizare:** Tailwind CSS - pentru un design responsive și modern.
- **Backend API:** Python (Flask) - servește ca proxy pentru Google Places API, gestionând cererile de geolocație și formatarea datelor.
- **Bază de date & Real-time:** Google Firebase Firestore - bază de date NoSQL pentru stocarea camerelor, utilizatorilor și sincronizarea voturilor în timp real.
- **Autentificare:** Firebase Authentication - implementare Google Sign-In.
- **API Extern:** Google Places API - pentru preluarea datelor reale despre restaurante (nume, poze, rating).

3 Cerințe funcționale

1. **Autentificare Google:** Utilizatorul trebuie să se poată loga rapid folosind contul de Google.
2. **Creare Cameră:** Utilizatorul poate crea o cameră nouă. Aplicația va cere permisiunea de localizare GPS și va interoga backend-ul Python pentru restaurantele din zonă.
3. **Join Room:** Alți utilizatori se pot alătura camerei folosind un cod unic generat (ex: "X7B2A").
4. **Mecanism de Swipe:** Utilizatorii pot vota "Nu" (Stânga) sau "Da" (Dreapta) pentru fiecare restaurant afișat.
5. **Real-time Matching:** Dacă toți utilizatorii din cameră votează pozitiv un restaurant, acesta apare instantaneu în lista de "Matches".

6. **Istoric:** Camerele create sau la care s-a participat sunt salvate în profilul utilizatorului pentru consultare ulterioară.
7. **Selecție Câștigător:** Când nu mai sunt restaurante de votat, aplicația alege aleatoriu un câștigător din lista de match-uri și oferă ruta către el.

3.1 Diagrama Use-Case

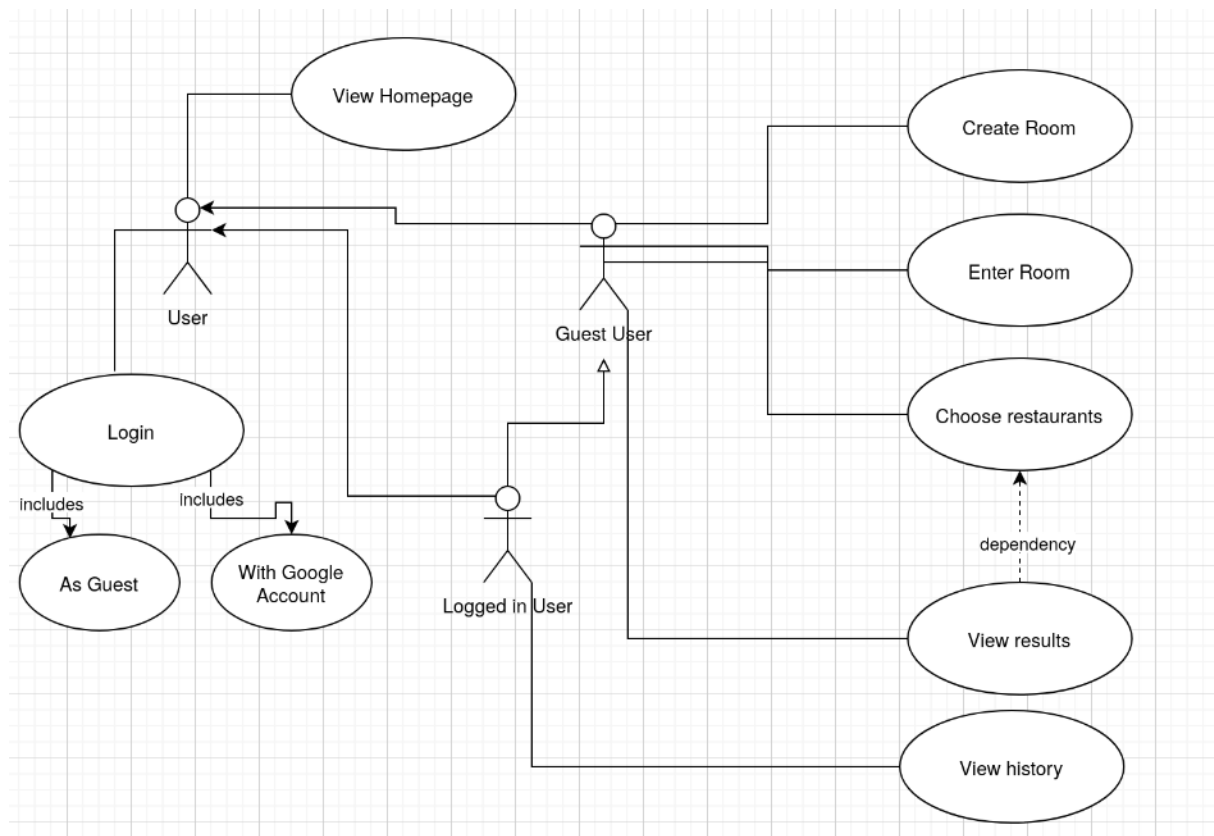


Figura 1: Diagrama Use-Case a sistemului TasteBuds

4 Cerințe non-funcționale

1. **Reactivitate:** Sincronizarea voturilor trebuie să se facă în timp real (latență sub 1 secundă) folosind WebSockets (via Firestore onSnapshot).
2. **Securitate:** Cheile API nu trebuie expuse în frontend; accesul la baza de date este restricționat pe baza ID-ului de utilizator.

3. **Scalabilitate:** Arhitectura trebuie să suporte multiple camere simultane fără degradarea performanței.
4. **Interfață Intuitivă:** Design-ul trebuie să fie minimalist, ușor de utilizat pe mobil ("Mobile First").

5 Design Patterns

Pentru a asigura un cod modular și ușor de întreținut, am implementat următoarele șabloane de proiectare. Fiecare pattern este documentat prin cele trei tipuri de diagrame (Communication, Sequence, State).

5.1 Factory Pattern

Implementat de: Cocoroadă Robert-Marius

Problema identificată: Crearea obiectului "Room" în React implica multă logică "boilerplate" dispersată (generarea ID-ului random, inițializarea structurilor de date pentru swipe-uri, setarea timestamp-ului). Acest lucru încălca principiul responsabilității unice.

Soluția: Am creat clasa `RoomFactory`. Aceasta centralizează logica de instanțiere. Metoda statică `create` primește doar datele esențiale (ID-ul utilizatorului și lista de restaurante) și returnează un obiect standardizat, gata de a fi salvat în baza de date.

Implementare:

```
class RoomFactory {
  static create(userId, restaurants) {
    const roomId = Math.random().toString(36).substring(2, 7).toUpperCase();
    return {
      id: roomId,
      users: [userId],
      restaurants: restaurants,
      swipes: { [userId]: {} },
      matches: [],
      created: new Date().toISOString(),
      status: 'active'
    };
  }
}
```

5.1.1 Diagrame Factory Pattern

5.2 Facade Pattern

Implementat de: Onofrei Ștefan Alexandru

Problema identificată: Gestionarea istoricului utilizatorului (salvarea camerelor și restaurantelor câștigate) implică multiple operații de scriere în baza de date Firebase și o structură complexă de colecții (users -> history). Expunerea acestei complexități direct în componentele React ar duce la o cuplare strânsă și la duplicarea logicii.

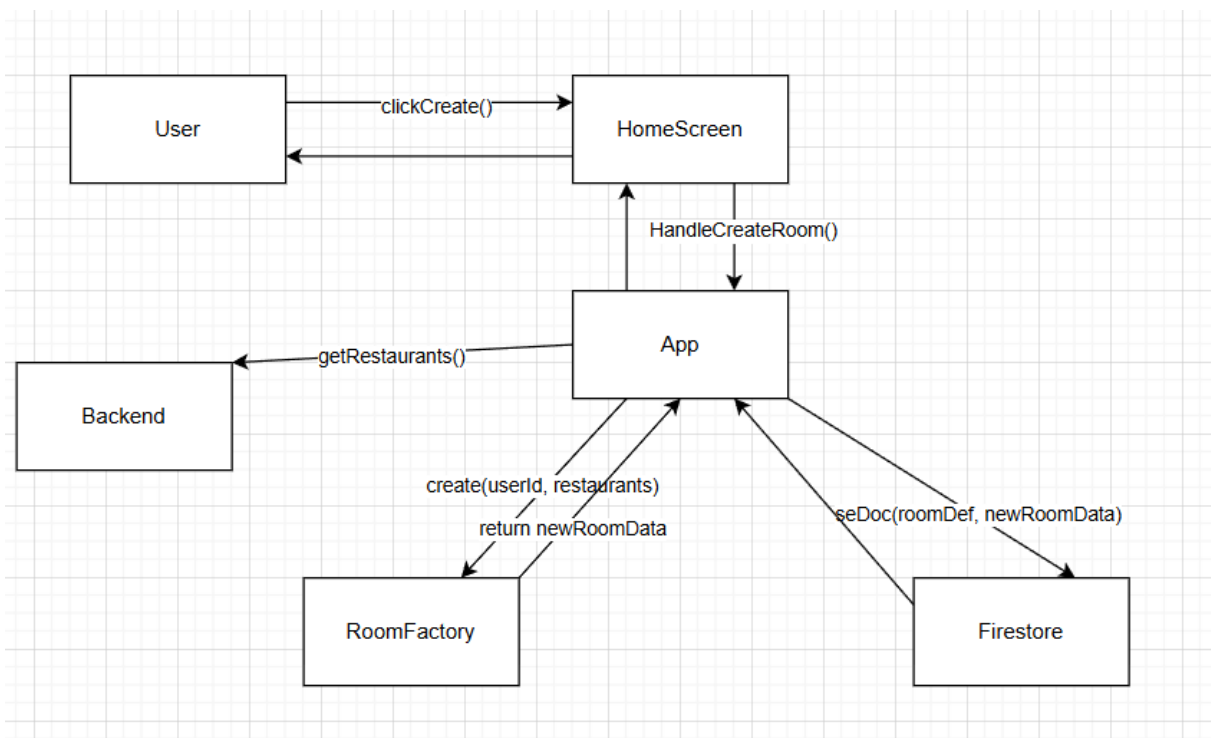


Figura 2: Diagrama de Comunicare - Factory Pattern

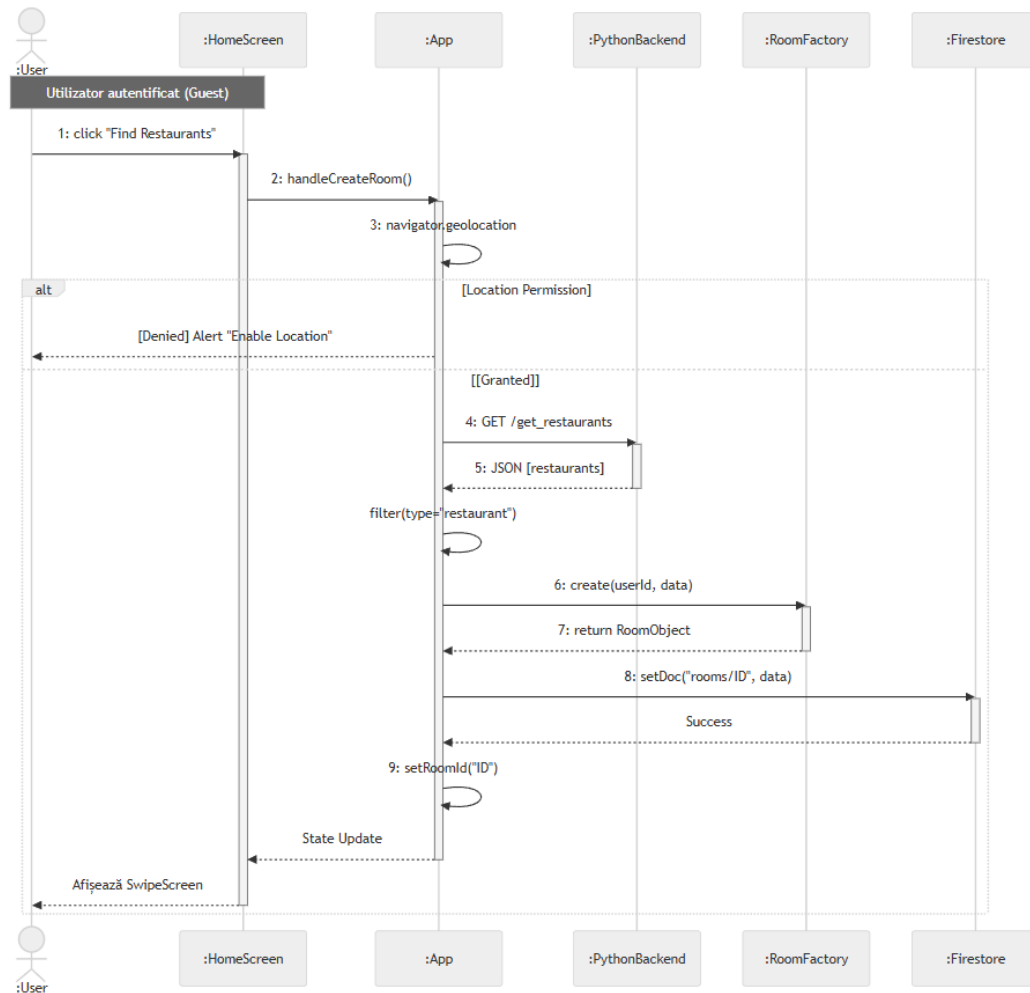


Figura 3: Diagrama de Secvență - Factory Pattern

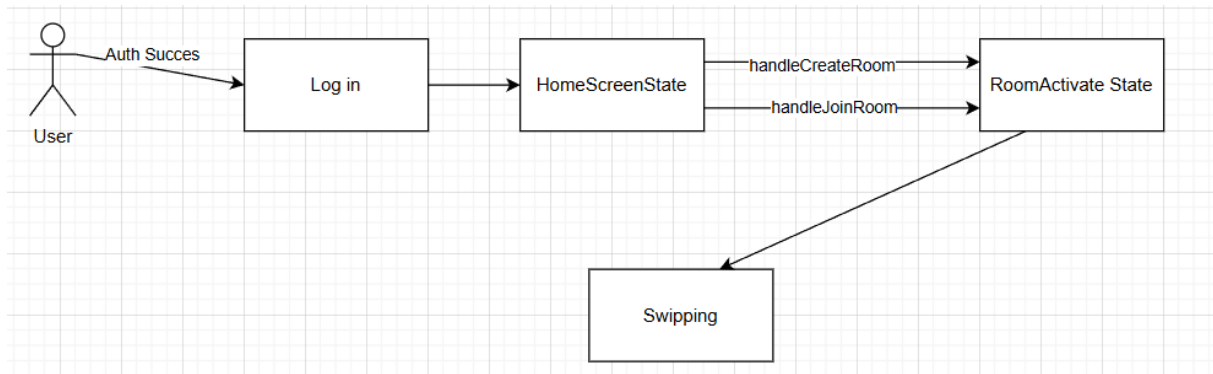


Figura 4: Diagrama de Stări - Factory Pattern

Soluția: Am implementat un `HistoryFacade` (în fișierul `services/historyService.js`). Acesta oferă o interfață simplificată pentru operațiile legate de istoric, ascunzând detaliile de implementare specifice Firestore. Componentele interacționează doar cu metodele fațadei, cum ar fi `saveToHistory`.

Implementare:

```
// services/historyService.js (Facade)
const HistoryFacade = {
  async saveToHistory(userId, roomData) {
    // Logica complexă de interacțiune cu Firestore este încapsulată aici
    try {
      const historyRef = doc(db, "users", userId, "history", roomData.id);
      await setDoc(historyRef, {
        roomId: roomData.id,
        date: new Date().toISOString(),
        winner: roomData.winner ? roomData.winner.name : "None"
      });
    } catch (error) {
      console.error("Failed to save history via Facade:", error);
    }
  }
};
export default HistoryFacade;
```

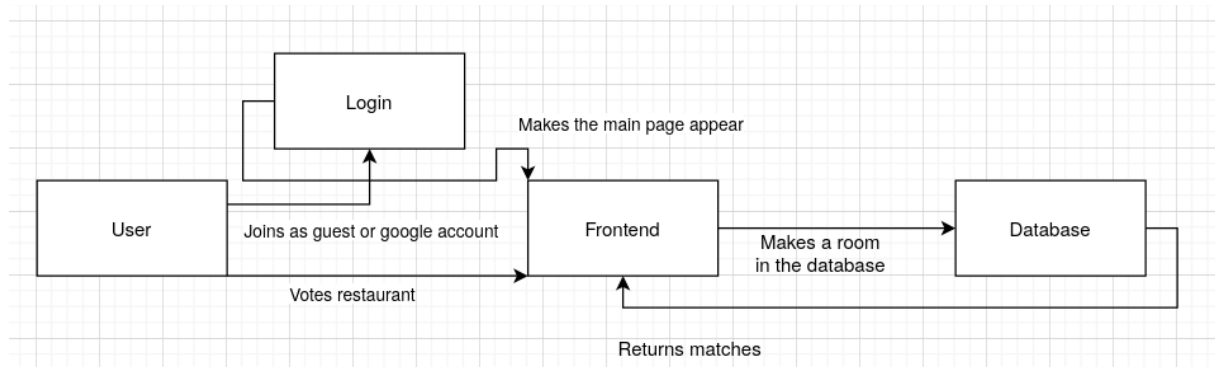



Figura 5: Diagrama de Comunicare - Facade Pattern

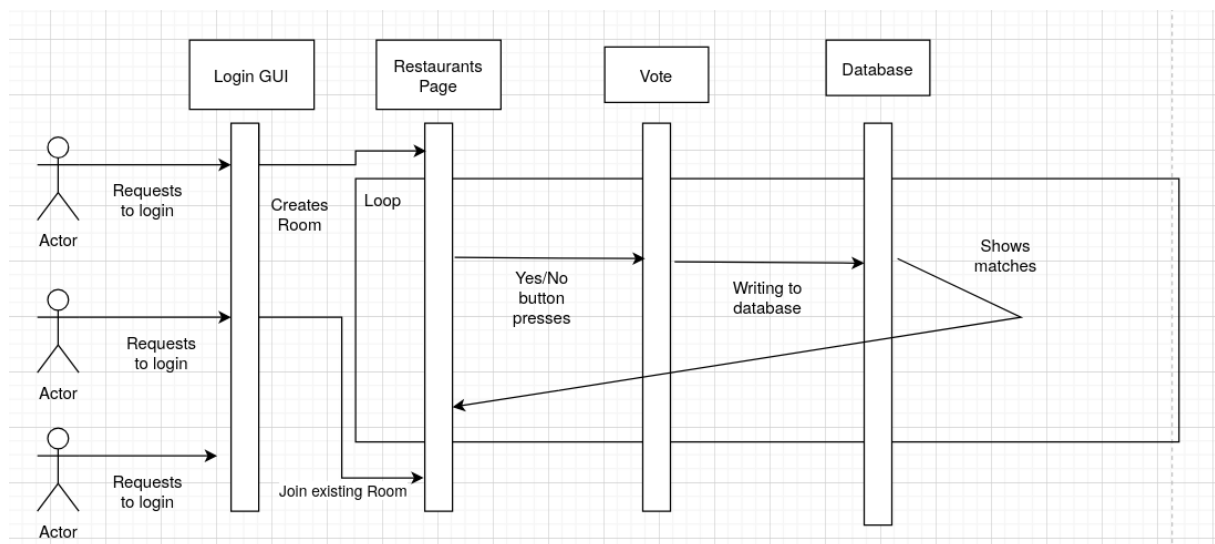


Figura 6: Diagrama de Secvență - Facade Pattern

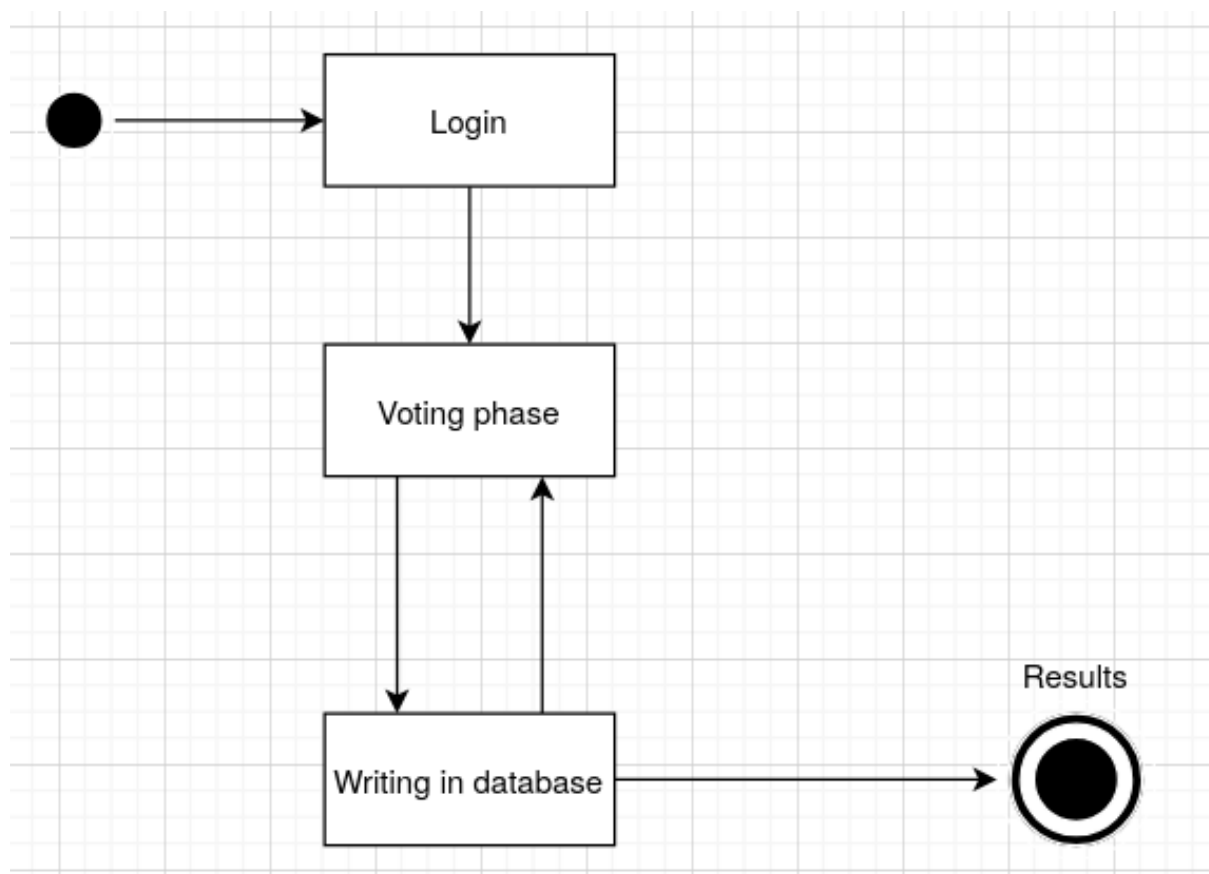


Figura 7: Diagrama de Stări - Facade Pattern

5.2.1 Diagrame Facade Pattern

6 Logică și Algoritmi Implementați

6.1 1. Geolocație și Integrare API

6.1.1 Descriere

Aplicația folosește API-ul nativ al browserului `navigator.geolocation` pentru a obține coordonatele (latitudine, longitudine) utilizatorului. Acestea sunt trimise către backend-ul Python, care interoghează Google Places API folosind o rază de 1500m.

6.1.2 Cod (Python Backend)

```
@app.get("/get_restaurants")
def get_restaurants():
    lat = request.args.get("lat")
    long = request.args.get("long")
    # ... apel catre Google Maps API ...
    return jsonify(formatted_restaurants)
```

6.2 2. Algoritmul de Matching

6.2.1 Descriere

Algoritmul de matching rulează la fiecare "Swipe Right" (Like). Acesta verifică dacă toți utilizatorii din cameră au votat pozitiv același restaurant. Este o intersecție a seturilor de voturi pozitive.

6.2.2 Pseudo-Cod

```
if (direction === "right") {
    // Verifică dacă ceilalți utilizatori au dat like
    const isMatch = otherUsers.every(user =>
        roomData.swipes[user.id][restaurantId] === "right"
    );

    if (isMatch) {
        addMatchToDatabase(restaurantId);
    }
}
```

6.3 3. Selecția Aleatorie a Câștigătorului

6.3.1 Descriere

Pentru a evita discuțiile interminabile chiar și după ce există mai multe opțiuni comune (Matches), aplicația include un algoritm simplu de decizie finală. Când lista de restaurante de parcurs s-a terminat, dacă există cel puțin un match, algoritmul selectează random un index din lista de match-uri.

6.3.2 Implementare

```
useEffect(() => {  
  if (!nextRestaurant && matches.length > 0 && !winner) {  
    const randomIndex = Math.floor(Math.random() * matches.length);  
    setWinner(matches[randomIndex]);  
  }  
}, [nextRestaurant, matches, winner]);
```

7 Instrucțiuni de Instalare (Read Me)

1. **Backend:** Se instalează dependențele (`pip install flask flask-cors`) și se pornește serverul: `python main.py`.
2. **Frontend:** Se instalează pachetele (`npm install`) și se pornește aplicația: `npm run dev`.
3. **Configurare:** Necesită un fișier `.env` cu cheile API pentru Firebase și Google Maps.