ExploreR — Generator de rute

Documentație proiect

Echipă: Razvan Valeanu, Emanuel Tutelea , Raul Negrea
An/Grupă: *Anul 3, grupa 7*
Disciplina: *Inginerie Software*

Cuprins

1 Scurtă descriere a proiectului **2**
1.1 Obiective 2

2 Limbaje și tehnologii folosite **2**

3 Cerințe funcționale + diagrama Use-Case **2**
3.1 Cerințe funcționale 2
3.2 Diagrama Use-Case 4

4 Cerințe non-funcționale **5**

5 Diagrame **6**
5.1 Razvan Valeanu 6
5.2 Emanuel Tutelea 9
5.3 Raul Negrea 11

6 Design Patterns **12**
6.1 Singleton — (Emanuel) 12
6.2 Middleware — (Raul) 12
6.3 Command — (Razvan) 13

7 Scurt README **13**
7.1 Structură proiect 13
7.2 Rulare locală (exemplu) 14
7.3 Funcții principale 14

1 Scurtă descriere a proiectului

ExploreR este o aplicație web care permite generarea automată de rute (pentru mașină, bicicletă sau pe jos) pe baza unor preferințe ale utilizatorului (zonă, distanță, punct de start). Utilizatorii pot salva rutele generate, le pot partaja într-un feed comun și pot interacționa social prin like-uri și comentarii.

1.1 Obiective

- Generarea de rute valide pe drumuri reale, cu parametri personalizabili.
- Persistența rutelor (salvare, detalii).
- Partajarea rutelor către un feed public/privat (în funcție de cerință).
- Interacțiuni sociale: like, comentariu, vizualizare detalii.

2 Limbaje și tehnologii folosite

- **Backend:** PHP (Laravel) — API REST, validări, persistență, autentificare.
- **Frontend:** Vue 3 + Vite — interfață, pagini, componente, integrare cu API.
- **Hartă / vizualizare:** Leaflet (sau echivalent) — randare traseu, marker-e, interacțiune.
- **Bază de date:** SQLite (dev) / *(opțional: MySQL/PostgreSQL în producție).*
- **Altele:** Git (versionare), JSON (schimb date), Postman/Insomnia (testare API).

3 Cerințe funcționale + diagrama Use-Case

3.1 Cerințe funcționale

1. **Autentificare**
   - Utilizatorul se poate autentifica pentru a accesa funcțiile de salvare/partajare/interacțiuni.
2. **Generare rută**
   - Utilizatorul selectează zona/punctul de start, distanța.
   - Sistemul generează o rută pe drumuri reale și o afișează pe hartă.
3. **Salvare rută**
   - Utilizatorul poate salva o rută generată (titlu, descriere, parametri, geometrie).
   - Sistemul generează/atribuie un *slug* unic pentru rută.
4. **Partajare rută**
   - Utilizatorul poate marca o rută ca partajată pentru a apărea în feed.
5. **Feed rute partajate**
   - Sistemul afișează o listă cu rute partajate (autor, dată, număr like-uri/comentarii).
   - Utilizatorul poate deschide detaliile unei rute din feed.
6. **Like / Unlike**

- Utilizatorul poate da like sau retrage like-ul de la o rută.
- Sistemul actualizează contorul de like-uri.

7. **Comentarii**

   - Utilizatorul poate vizualiza comentariile unei rute și poate adăuga un comentariu nou.
   - Sistemul actualizează contorul de comentarii.

8. **Vizualizare detalii rută**

   - Sistemul afișează geometria rutei pe hartă și metadate (distanță, timp estimat, etc. dacă există).

3.2 Diagrama Use-Case

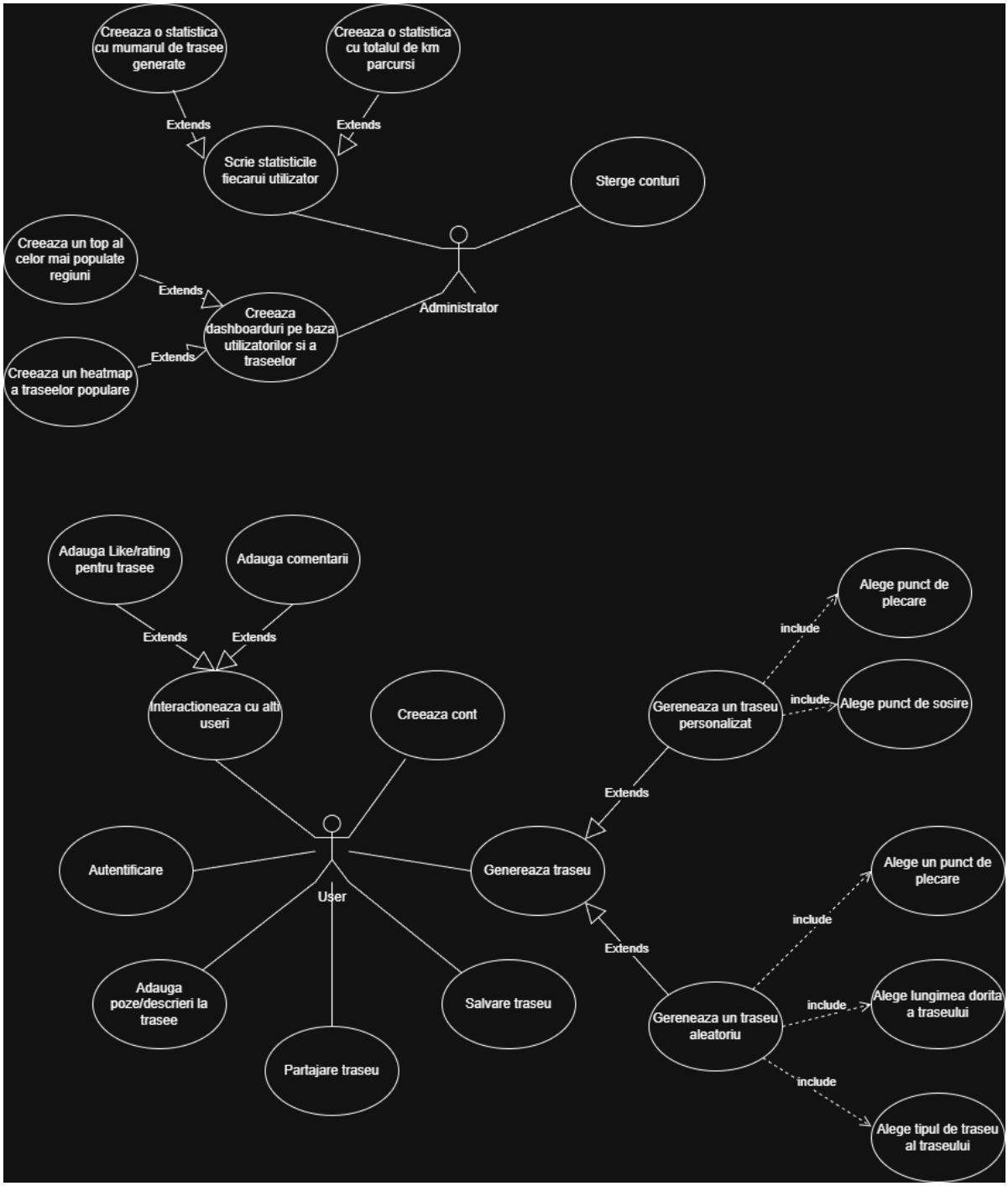

Figura 1: Diagrama Use-Case — ExploreR

4 Cerințe non-funcționale

- **Performanță:** afișarea feed-ului și a detaliilor rutei trebuie să fie responsivă; cererile API uzuale sub 1s în mediu local.
- **Fiabilitate:** datele salvate (rute, like-uri, comentarii) trebuie persistate corect și coerent.
- **Securitate:** endpoint-urile de like/comentariu/salvare necesită utilizator autentificat; validare input (lungime text, tipuri).
- **Ușurință în utilizare:** UI clar (butoane Generate/Save/Share/Like/Comment), feedback vizual la acțiuni.
- **Scalabilitate:** structura permite migrarea de la SQLite la o bază de date server fără rescrieri majore.
- **Mentenabilitate:** separare pe straturi (frontend componente, backend controllers/services/models), cod modular.

5 Diagrame

5.1 Razvan Valeanu

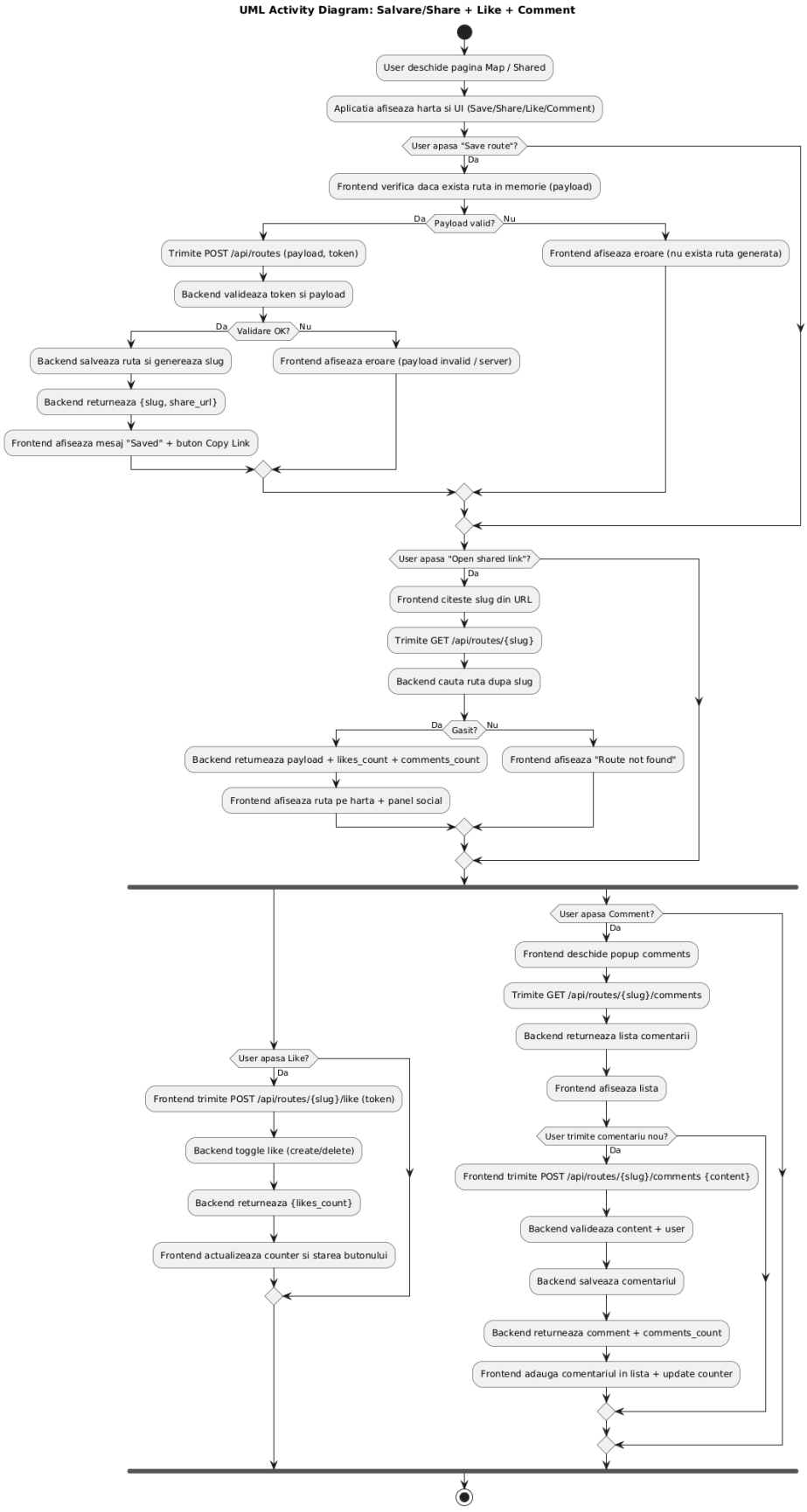

Figura 2: Activity Diagram — (Razvan Valeanu)

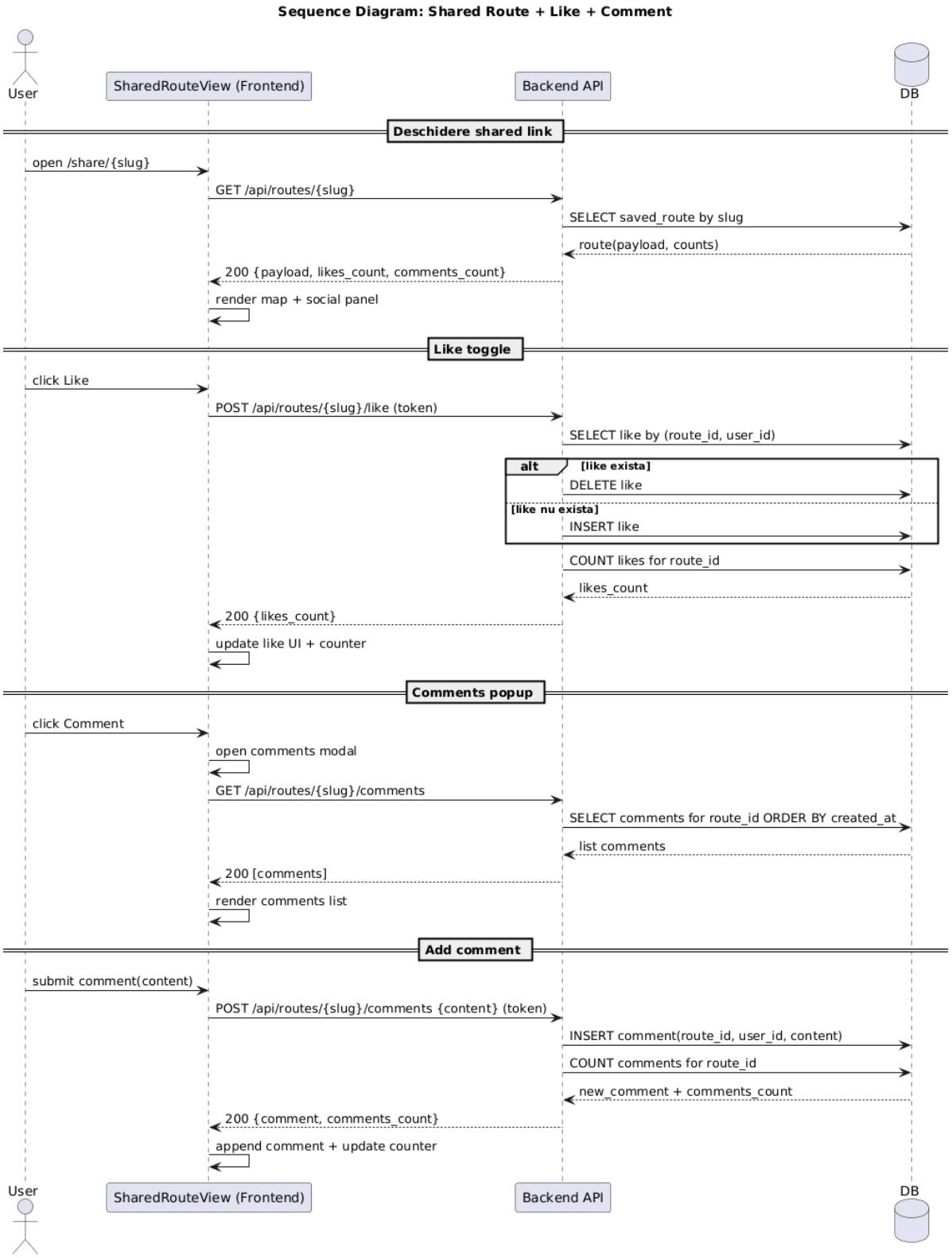

Figura 3: Sequence Diagram — (Razvan Valeanu)

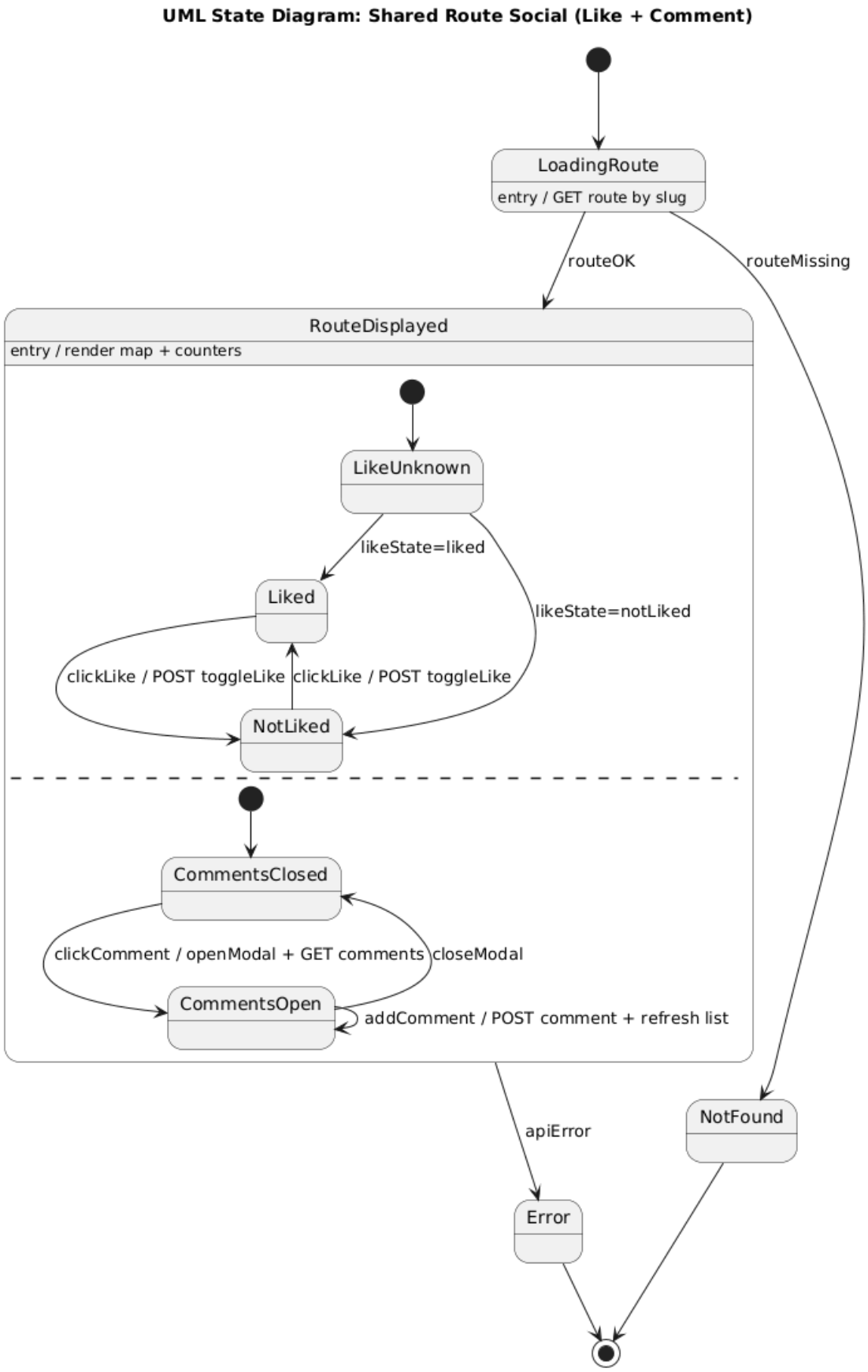

Figura 4: State Diagram — (Razvan Valeanu)

5.2 Emanuel Tutelea

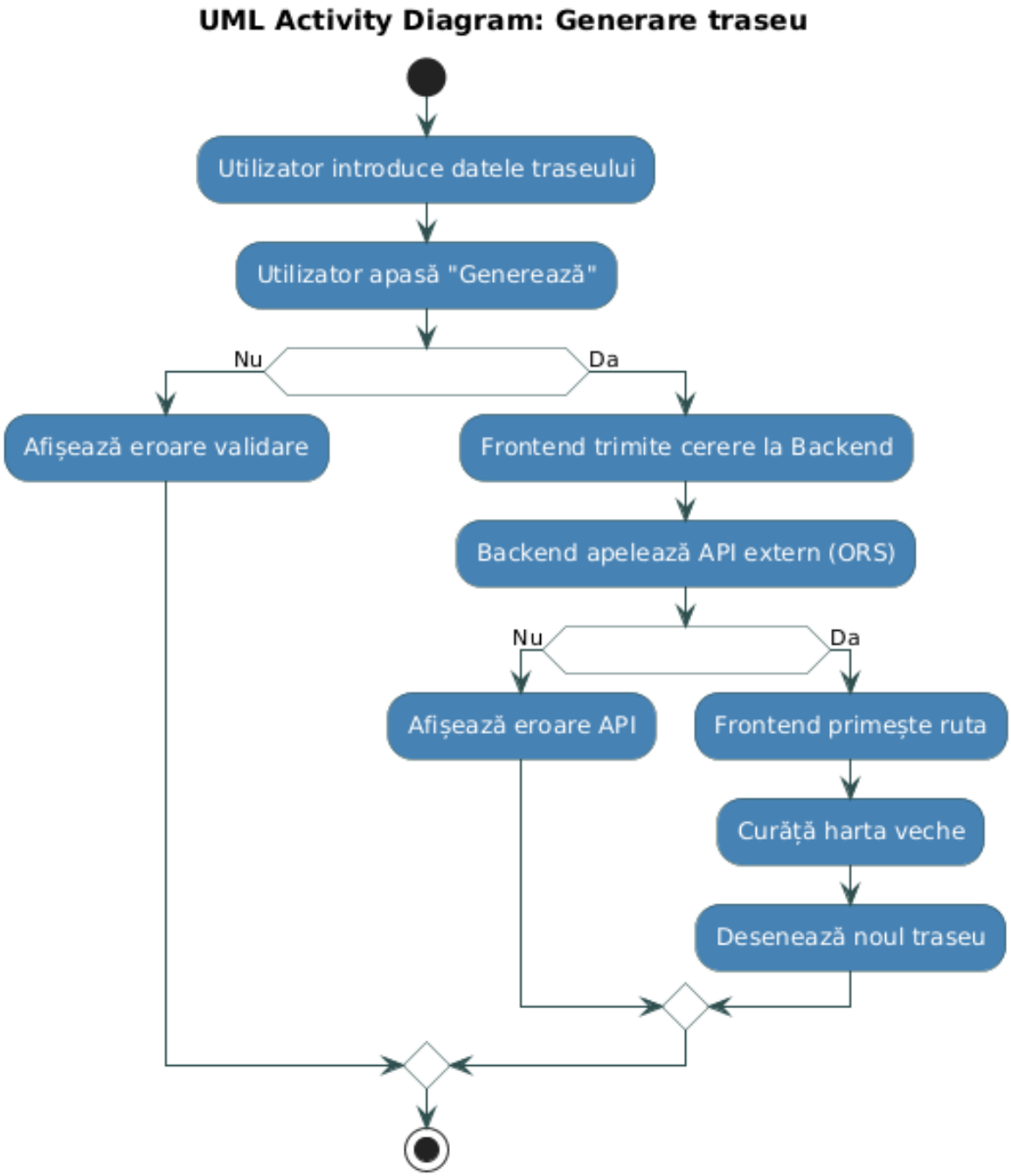

Figura 5: Activity Diagram — (Emanuel)

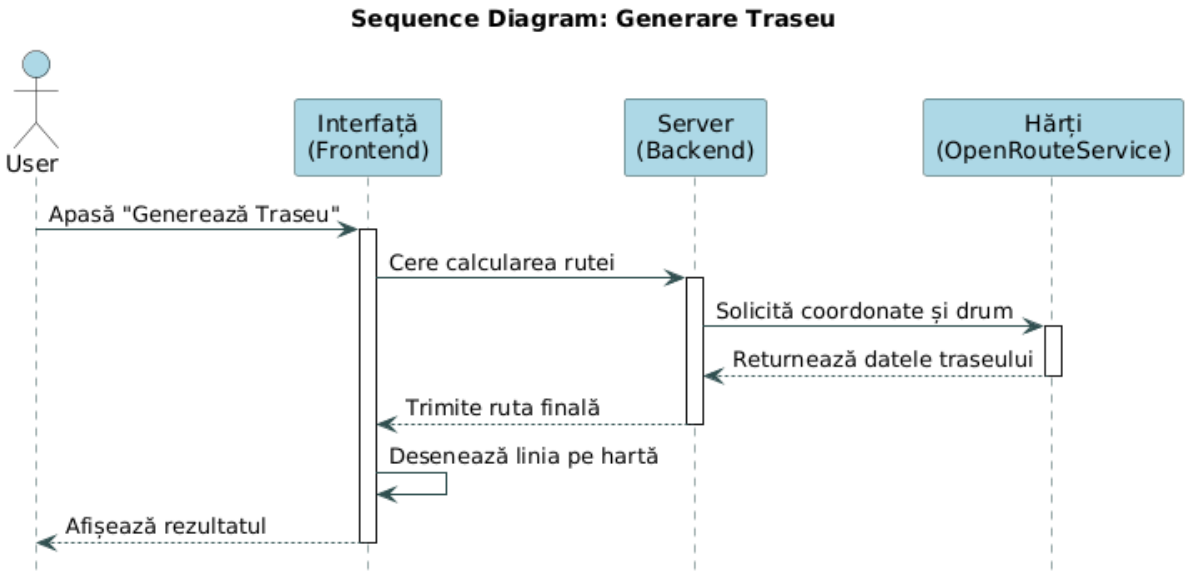

Figura 6: Sequence Diagram — (Emanuel)

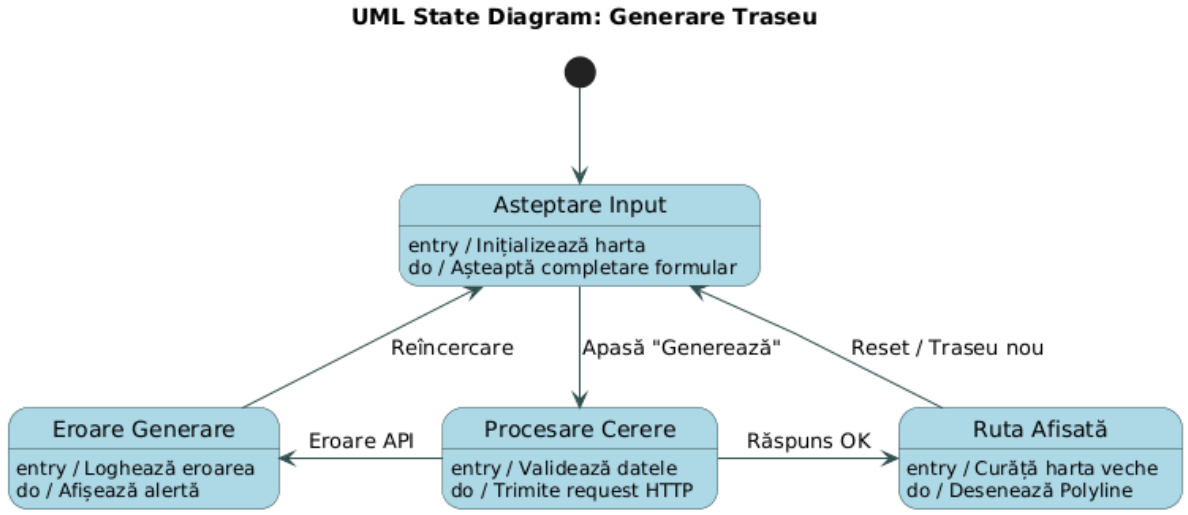

Figura 7: State Diagram — (Emanuel)

5.3 Raul Negrea

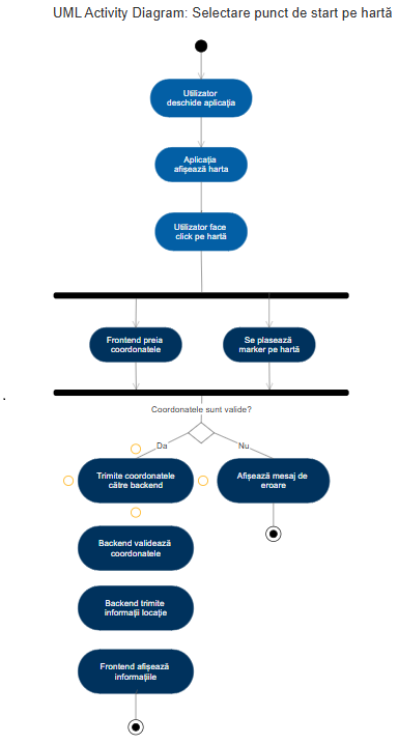

Figura 8: Activity Diagram — (Raul)

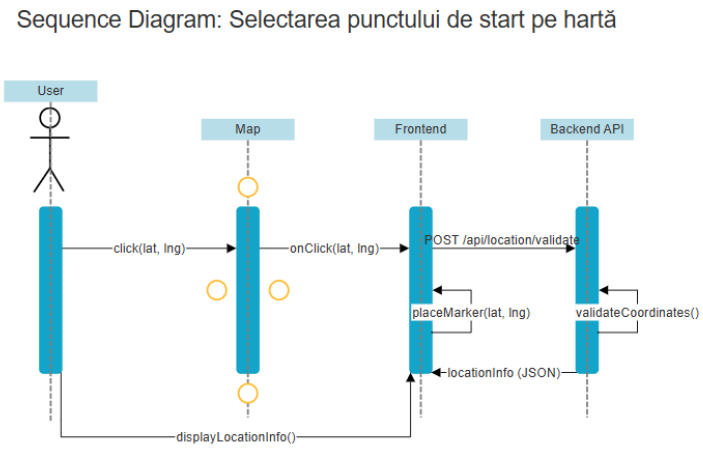

Figura 9: Sequence Diagram — (Raul)

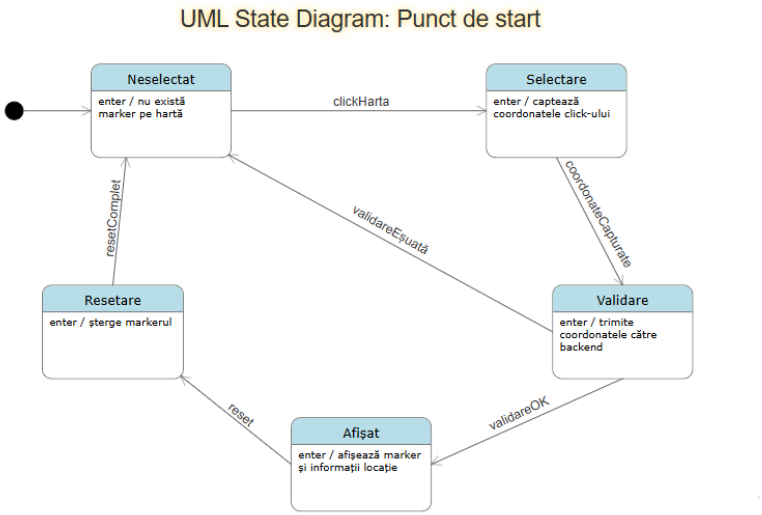

Figura 10: State Diagram — (Raul)

6 Design Patterns

6.1 Singleton — (Emanuel)

Problema identificată: Aplicația folosește serviciul OpenRouteService și are nevoie de cheia API în mai multe rute. Citirea repetată a acestei chei din fișierul de configurare (.env) la fiecare request este ineficientă.

Soluția (Singleton): Am implementat o clasă cu instanță unică care citește cheia o singură dată și o păstrează în memorie pe durata execuției, oferind un punct de acces global.

Detalii legate de implementare:

- Am creat clasa KeyManager în folderul Providers, având constructorul privat.
- Cheia este încărcată o singură dată din .env la prima utilizare.
- În restul aplicației (în web.php), cheia se obține doar prin apelul: KeyManager::getInstance() ->getKey().

6.2 Middleware — (Raul)

Problema identificată: Unele endpoint-uri (ex.: salvare rute, listare rute salvate, ștergere) trebuie accesate **doar de utilizatori autentificați**. Dacă verificarea token-ului este făcută direct în fiecare controller / endpoint, apare duplicare de cod, logică împrăștiată și risc de inconsistențe (un endpoint uitat neprotejat).

Soluția (Middleware): Se introduce un middleware care interceptează request-urile **înainte** să ajungă la controller și face verificarea autentificării (ex.: header `Authorization: Bearer <token>`). Dacă token-ul este valid, request-ul continuă; dacă nu, se oprește cu `401 Unauthorized`. Astfel, controller-ele rămân curate și se ocupă strict de business logic.

Detalii legate de implementare:

- Se creează middleware-ul `ApiTokenAuth` cu metoda `handle(request, next)`.
- Middleware-ul:
  - citește header-ul `Authorization`;

  - validează formatul `Bearer`;
  - decodează token-ul și identifică utilizatorul;
  - dacă utilizatorul nu e valid → răspunde cu `401`;
  - altfel → apelează `next(request)`.

- Endpoint-urile protejate sunt grupate cu middleware:

```
Route::middleware('api.token')->group(function () {
  // ex: POST /routes, GET /routes, DELETE /routes/{slug}
});
```

- Endpoint-urile publice (ex.: `GET /routes/{slug}` pentru share link) rămân în afara grupului, deci pot fi accesate fără login.

Beneficiu direct: autentificarea este centralizată într-un singur loc, controller-ele nu mai conțin verificări repetitive, iar adăugarea de reguli noi (ex.: roluri, rate-limit, verificare expirare token) se face prin modificarea/compunerea middleware-urilor, fără să atingi logica endpoint-urilor.

6.3 Command — (Razvan)

Problema identificată: Interacțiunile sociale (ex.: `toggleLike`, adăugare comentariu) introduc logică de business în controller (`RouteSocialController`). Pe măsură ce apar mai multe acțiuni (edit/delete comment, report, etc.), controller-ul se umple cu cod repetitiv și devine greu de extins și testat.

Soluția (Command): Fiecare acțiune socială este modelată ca o comandă separată, cu o singură responsabilitate (ex.: `ToggleLikeCommand`, `AddCommentCommand`). Controller-ul doar colectează datele necesare (user, rută, payload) și execută comanda corespunzătoare, apoi returnează rezultatul.

Detalii legate de implementare:

- Se definește o interfață comună pentru comenzi (ex.: `SocialCommand`) cu metoda `execute(user, route, payload)`.
- Logica existentă din `RouteSocialController::toggleLike()` este mutată într-o clasă dedicată `ToggleLikeCommand`:
  - verifică dacă există deja un like (`RouteLike` pentru `route_id` + `user_id`);
  - face `delete` dacă există, altfel `create`;
  - recalculează `likes_count` și îl returnează către UI.
- Controller-ul rămâne „thin”: caută ruta după `slug`, ia utilizatorul curent și cheamă comanda (ex.: `$cmd->execute($user, $route)`), apoi returnează JSON-ul (ex.: `{"likes_count":...}`).
- Extinderea pentru comentarii se face prin comenzi noi (ex.: `AddCommentCommand`, `ListCommentsCommand`), fără modificări majore în controller.

7 Scurt README

7.1 Structură proiect

- `/backend` — Laravel API (auth, rute, like, comentarii, feed)
- `/frontend` — Vue + Vite (UI, map, feed, route details)

7.2 Rulare locală (exemplu)

Backend (Laravel):

1. `cd backend`
2. `cp .env.example .env` (dacă nu există `.env`)
3. `composer install`
4. `php artisan key:generate`
5. `php artisan migrate`
6. `php artisan serve`

Frontend (Vue + Vite):

1. `cd frontend`
2. `npm install`
3. `npm run dev`

7.3 Funcții principale

- Generare rută + afișare pe hartă
- Salvare rută (cu slug)
- Partajare în feed
- Like/unlike + comentarii (cu actualizare UI)