

1. INTRODUCERE

1.1 Scopul principal

Împreună cu îndrumarea coordonatorului științific Ș. L. Dr. Ing. Cerbulescu Cătălin și la sugestia companiei NetRom Software, a fost aleasă ca temă de licență dezvoltarea unui proiect de tip ride-sharing.

FastRide este o aplicație de ride-sharing, concepută pentru a facilita legătura directă între clienți și șoferi, într-un mod rapid și eficient.

Scopul principal al aplicației FastRide este de a oferi un mediu online prin care utilizatorii (clienți) pot cere curse în timp real, iar șoferii disponibili le pot accepta într-un mod simplu și intuitiv. Aplicația urmărește să automatizeze complet procesul de conectare dintre cererea și oferta de transport urban, eliminând nevoia de apeluri telefonice.

1.2 Motivație

Tema aleasă pentru această lucrare pornește de la nevoia pentru accesul rapid și eficient la servicii de transport. Odată cu dezvoltarea orașelor și creșterea traficului, s-a observat o tendință accentuată spre utilizarea aplicațiilor de tip ride-sharing, care oferă o alternativă flexibilă la transportul clasic.

Aplicația FastRide propune o soluție simplă și eficientă pentru gestionarea curselor între clienți și șoferi. Ideea de a dezvolta această aplicație a venit din dorința de a construi un proiect complet, în care să se regăsească elemente din tot ce înseamnă dezvoltare software modernă: o interfață prietenoasă, comunicare în timp real, stocare în cloud și integrare cu servicii externe.

Tehnologiile alese: Blazor WebAssembly pentru partea de front-end, Azure Durable Functions pentru back-end și SignalR pentru actualizări live, au permis realizarea unei aplicații distribuite, capabile să răspundă în timp real cerințelor utilizatorilor. De asemenea, integrarea cu Stripe pentru validarea cardurilor și autentificarea prin cont Google au contribuit la crearea unei experiențe cât mai fluide și sigure.

Prin aceasta lucrare mi-am propus dezvoltarea unei aplicații funcționale, care poate fi ușor extinsă și adaptată în viitor, dar și învățarea unor tehnologii moderne folosite în proiecte reale.

2. TEHNOLOGII ȘI FRAMEWORK-URI FOLOSITE

2.1 C#

C# este un limbaj de programare modern, orientat pe obiect, dezvoltat de Microsoft și lansat în anul 2000, ca parte a inițiativei .NET. A fost creat sub conducerea lui Anders Hejlsberg, același care a contribuit la dezvoltarea limbajului Delphi. Scopul principal al C# a fost să combine puterea și flexibilitatea limbajelor precum C++ cu simplitatea și siguranța limbajelor moderne, cum ar fi Java. [9]

C# este un limbaj conceput pentru a fi ușor de învățat și folosit, dar suficient de robust pentru aplicații complexe. Este ideal pentru dezvoltarea de aplicații desktop, web, mobile, jocuri (cu Unity), dar și servicii cloud, microservicii și funcții serverless, în special atunci când este combinat cu platforma .NET. [9]

Limbajul C# este orientat pe obiect, adică permite organizarea codului în clase și obiecte, facilitând reutilizarea și întreținerea codului. De asemenea, are suport nativ pentru asincronism (prin *async/await*), LINQ pentru interogarea datelor într-un mod elegant și declarativ, și pattern matching pentru expresivitate crescută. [9]

C# a evoluat constant, adăugând funcționalități care îl fac competitiv cu cele mai populare limbaje de programare. Ultimele versiuni aduc îmbunătățiri semnificative în ceea ce privește performanța, claritatea sintaxei și siguranța codului. [9]

Este adesea folosit în mediul enterprise datorită integrării excelente cu serviciile Microsoft (Azure, SQL Server, Active Directory etc.) și a ecosistemului .NET, care oferă o platformă stabilă, multiplatformă și bine întreținută. [9]

În dezvoltarea aplicațiilor C#, pentru organizarea codului există conceptele de design patterns, care oferă soluții reutilizabile pentru probleme comune de arhitectură software. [9]

Una dintre cele mai utilizate structuri este separarea aplicației în straturi distincte: *Repository* și *Service*. Acest model este specific abordărilor precum Clean Architecture și Domain-Driven Design. [3]

Repository-ul este responsabil pentru interacțiunea cu sursa de date, fie că este o bază de date relațională, un storage NoSQL sau un API extern. El oferă o interfață clară pentru operații precum adăugarea, citirea, actualizarea sau ștergerea entităților, abstractizând detaliile concrete de stocare. [3]

Service-ul vine deasupra repository-ului și conține logica de business a aplicației. Aici sunt implementate regulile și fluxurile aplicației, validările și transformările de date. Service-ul folosește repository-ul ca sursă de date, dar nu se ocupă direct cu modul în care acestea sunt stocate sau obținute. [3]

2.2 .NET

.NET este o platformă de dezvoltare creată de Microsoft, concepută pentru a permite dezvoltatorilor să construiască aplicații performante pentru o varietate largă de dispozitive și sisteme. A fost lansată oficial în anul 2002 sub numele de .NET Framework, fiind destinată exclusiv sistemelor Windows. Scopul principal era de a oferi un cadru unitar, stabil și coerent în care dezvoltatorii să poată crea aplicații desktop și web într-un mod eficient, folosind limbaje precum C# și VB.NET. [10]

Pe măsură ce tehnologia a evoluat, Microsoft a început procesul de modernizare a platformei, orientându-se către dezvoltarea multiplatformă și open-source. Astfel, în 2016 a fost lansat .NET Core, o versiune rescrisă și modulară a platformei, capabilă să ruleze pe Windows, Linux și macOS. Aceasta a oferit o arhitectură mai flexibilă și un sistem de distribuție mai eficient, prin intermediul pachetelor NuGet. [10]

Închind cu .NET 5, lansat în 2020, Microsoft a unificat direcțiile .NET Framework și .NET Core într-o singură platformă, denumită simplu .NET. Această versiune unificată continuă să evolueze, cu îmbunătățiri constante aduse în performanță, suportul pentru limbaje, instrumente de dezvoltare și capabilități cloud. Versiunile recente, cum ar fi .NET 6, .NET 7 și .NET 8, sunt toate versiuni LTS (Long Term Support) sau standard, folosite pe scară largă în industrie. [10]

.NET se bazează pe concepte precum CLR (Common Language Runtime), care este motorul de execuție ce gestionează rularea aplicațiilor, și BCL (Base Class Library), o colecție vastă de librării care oferă funcționalități esențiale: lucrul cu fișiere, colecții, rețea, criptare, baze de date și multe altele. De asemenea, platforma oferă suport excelent pentru programarea asincronă, gestionarea memoriei automate prin Garbage Collection, precum și o integrare nativă cu servicii și tehnologii moderne precum Azure, Docker și Kubernetes. [10]

Prin intermediul .NET, dezvoltatorii pot crea aplicații web (cu ASP.NET), desktop (cu WPF și WinForms), mobile (cu .NET MAUI și Xamarin), servicii API, aplicații cloud, jocuri (prin Unity) și chiar soluții de inteligență artificială sau machine learning. [10]

2.3 Azure Function

Azure Functions este o platformă serverless dezvoltată de Microsoft care permite rularea codului în cloud fără a fi nevoie să gestionezi infrastructura de servere. Acest model facilitează dezvoltarea rapidă a aplicațiilor și serviciilor scalabile, concentrându-se doar pe logică, nu pe administrarea resurselor. [15]

Principalele caracteristici ale Azure Functions sunt:

- Execuție event-driven: Funcțiile sunt declanșate automat de evenimente, cum ar fi modificări în baza de date, mesaje din cozi, cereri HTTP, cronometre sau alte surse.

- Scalabilitate automată: Azure gestionează în mod automat scalarea funcțiilor în funcție de cerere, asigurând performanță optimă indiferent de volumul de trafic.
- Model de plată pay-as-you-go: Se plătește doar pentru timpul efectiv în care codul rulează, fără costuri fixe legate de infrastructură.
- Suport pentru mai multe limbaje: C#, JavaScript, Python, Java, PowerShell și altele pot fi folosite pentru a scrie funcțiile.

[15]

2.3.1 Azure Durable Functions

Azure Durable Functions reprezintă o extensie a platformei Azure Functions, dezvoltată de Microsoft pentru a facilita crearea de fluxuri de lucru pe termen lung și orchestrarea funcțiilor serverless într-un mod eficient și scalabil. Lansată oficial în 2017, această extensie adaugă capabilități suplimentare funcțiilor serverless tradiționale, permițând dezvoltatorilor să gestioneze procese complexe care implică mai multe funcții interdependente și executate în timp. [8]

Azure Functions, lansat cu un an mai devreme (2016), face parte din suita de servicii serverless computing din Azure și permite rularea de cod fără a fi necesară administrarea explicită a infrastructurii. Totuși, aceste funcții standard au fost concepute pentru execuții rapide, de scurtă durată, ceea ce le făcea mai puțin potrivite pentru procesele care necesită menținerea stării și coordonarea pe termen lung. [8]

Pentru a răspunde acestor nevoi, Durable Functions oferă următoarele capabilități:

- Orchestrarea funcțiilor de scurtă durată, într-un mod automatizat și declarativ;
- Gestionarea stării între apelurile funcțiilor, pe parcursul execuției unui proces complex;
- Retry automat și suport pentru scenarii de compensare în caz de eșec.

[8]

Durable Functions este construit pe baza Durable Task Framework, permițând dezvoltatorilor să scrie cod orchestrat într-un stil secvențial, dar care este transformat automat în execuție asincronă și distribuită, cu păstrarea stării între pași. [8]

Scenarii comune de utilizare:

- Orchestrarea fluxurilor de lucru - o funcție orchestrator coordonează apelurile către alte funcții, în funcție de anumite condiții sau răspunsuri. De exemplu, poate apela o funcție care extrage date de la un API, apoi, în funcție de rezultat, lansează alte funcții în lanț.
- Function Chaining (lanțuri de funcții) - mai multe funcții sunt apelate secvențial, iar rezultatul fiecărei funcții este transmis mai departe către următoarea funcție din lanț.

- Fan-out/Fan-in - o funcție poate declanșa mai multe funcții în paralel (fan-out), după care agregă rezultatele într-un punct comun (fan-in). Acest model este ideal pentru procesarea paralelă a unor seturi mari de date.
- Gestionarea proceselor de lungă durată - Durable Functions permite execuția de fluxuri care pot dura ore, zile sau chiar luni, menținând starea între pași. Este util în scenarii precum aprobări, procese de onboarding, migrare de date sau procese distribuite în timp.
- Compensarea acțiunilor (Saga Pattern) - În fluxuri unde mai multe acțiuni trebuie executate într-o ordine strictă, Durable Functions permite implementarea de logici de rollback sau compensare în caz de eșec, asigurând consistența procesului.
- Funcții temporizate (Timer Functions) - Orchestratorii pot programa funcții să ruleze după o întârziere sau la anumite intervale. Acestea sunt utile în procese automate, monitorizări periodice sau trimiterea de notificări programate.

[8]

Lucrarea folosește Azure Functions, în special Azure Durable Functions pentru a face posibilă procesarea și menținerea legăturii între 2 utilizatori pe parcursul cursei, indiferent de circumstanțele actuale (utilizatorul face Refresh sau aplicația își ia Restart).

2.4 Azure Table Storage

Azure Table Storage este un serviciu creat de Microsoft ca parte a suitei Azure Storage, gândit pentru a oferi o soluție simplă, scalabilă și extrem de rapidă pentru stocarea datelor structurate în format NoSQL. Table Storage nu impune o schemă fixă, ceea ce înseamnă că fiecare entitate dintr-o tabelă poate avea un set diferit de proprietăți. Acest lucru îl face ideal pentru scenarii în care datele sunt variabile sau când flexibilitatea contează mai mult decât relațiile complexe dintre entități. [16]

A fost introdus în jurul anului 2010, într-o perioadă în care Microsoft își contura strategia cloud și începea să ofere servicii care să concureze cu AWS. A fost conceput pentru a servi aplicații la scară mare, care necesită o cantitate mare de scrieri și citiri, dar fără nevoia de tranzacții complexe sau relații între tabele. Scenariile clasice includ loguri, telemetrie, date despre utilizatori, mesaje sau orice tip de evenimente care trebuie stocate rapid și recuperate eficient. [16]

Table Storage folosește o structură bazată pe tabele care conțin entități, fiecare identificată printr-o combinație unică de *PartitionKey* și *RowKey*. Această combinație asigură performanță crescută la căutare, mai ales atunci când datele sunt distribuite corect în funcție de *PartitionKey*. Fiind un serviciu NoSQL, nu există conceptul de join-uri sau constrângeri între entități, dar în schimb se câștigă foarte mult la capitolul viteză și scalabilitate. [16]

În timp, Table Storage a devenit o alegere populară pentru aplicațiile moderne, în special cele distribuite, microservicii sau orice alt sistem care are nevoie de o bază de date simplă, ieftină și elastică. Este integrat profund în ecosistemul Azure, ceea ce înseamnă că poate fi combinat ușor cu alte servicii precum Azure Functions, Logic Apps sau Azure Event Grid, făcându-l extrem de versatil în arhitecturi cloud-native. [16]

Azure Storage reprezintă structura de stocare pentru lucrarea de față, datorită numărului scăzut de dependențe și tabele utilizate și vitezei de care dispune.

2.4.1 Azurite

Azurite este un emulator local creat de Microsoft pentru serviciile de stocare din Azure. A fost conceput ca un instrument de dezvoltare care le permite programatorilor să lucreze cu Azure Blob Storage, Queue Storage și Table Storage pe propriul calculator, fără a avea nevoie de o conexiune activă la platforma Azure. Practic, simulează comportamentul serviciilor reale, oferind o experiență de dezvoltare aproape identică cu mediul de producție. [14]

A apărut ca succesor al emulatorului mai vechi Microsoft Azure Storage Emulator, care era limitat la Windows. Azurite, în schimb, este cross-platform, fiind scris în Node.js și disponibil ca pachet NPM sau container Docker. Asta îl face ideal pentru proiecte care rulează pe macOS, Linux sau Windows, fie în linie de comandă, fie integrat în Visual Studio Code. [14]

Oferă un mediu de testare local rapid și fără costuri, în care dezvoltatorii pot simula citiri, scrieri, partajări și alerte, fără riscul de a consuma resurse reale în Azure sau de a introduce erori în datele live. Este frecvent folosit în timpul dezvoltării și al testării automate, mai ales în scenarii unde aplicația interacționează frecvent cu Table Storage sau Blob-uri. [14]

Azurite ajută la accelerarea dezvoltării, reduce costurile și elimină nevoia de conexiune constantă la cloud, păstrând în același timp consistența cu ceea ce se întâmplă în Azure real. Din acest motiv este folosit și în proiectul curent.

2.5 LINQ

LINQ (Language Integrated Query) este o componentă a limbajului C# care permite interogarea colecțiilor de date într-un mod declarat, expresiv și tip-safe, folosind o sintaxă integrată direct în limbaj. LINQ oferă o punte între programarea orientată pe obiecte și interogarea datelor, permițând manipularea colecțiilor precum liste, array-uri, baze de date sau fișiere XML. [11]

Prin LINQ, sintaxa este similară celei din SQL, dar adaptată paradigmei C# și poate fi aplicată atât în memorie (de exemplu, asupra listelor), cât și asupra surselor externe, cum ar fi Entity Framework sau XML. [11]

În contextul Azure Table Storage, LINQ poate fi folosit pentru a interoga datele stocate într-un mod expresiv, dar trebuie menționat că suportul LINQ este mai limitat comparativ cu

interogările LINQ în memorie sau cu Entity Framework. Azure Table Storage este un sistem NoSQL key-value, iar interogările se bazează pe filtrarea entităților în funcție de *PartitionKey*, *RowKey* și alte proprietăți. [11]

Pentru a lucra cu LINQ în Azure Table Storage, se folosește de regulă SDK-ul oficial (*Azure.Data.Tables*) și clase ca *TableClient* sau *TableServiceClient*. [11]

Utilizarea LINQ peste *TableQueryable<T>* (în SDK-uri mai vechi) sau direct cu *TableClient.Query<T>()* în SDK-ul modern, există access la o subset de metode LINQ compatibile cu server-side query translation. Cele mai importante metode disponibile sunt:

- *Where()* – pentru a filtra datele după proprietăți simple
- *Take()* – pentru a limita numărul de rezultate returnate
- *Select()* – parțial suportat, pentru a proiecta obiectele într-o formă nouă
- *OrderBy()* și *OrderByDescending()* – doar în anumite condiții și nu întotdeauna disponibile, în funcție de SDK

[11]

2.6 HyperText Markup Language (HTML)

HTML, prescurtarea de la HyperText Markup Language, este limbajul standard folosit pentru a structura și a afișa conținut pe web. A apărut la începutul anilor '90, dezvoltat de Tim Berners-Lee, și a stat la baza creării primelor pagini web din internetul modern. HTML nu este un limbaj de programare propriu-zis, ci un limbaj de marcare, ceea ce înseamnă că este folosit pentru a indica browserului cum să interpreteze și să afișeze diverse elemente: titluri, paragrafe, imagini, linkuri, liste, formulare și multe altele. [2]

Documentele HTML sunt compuse din etichete (tag-uri) care înconjoară conținutul pentru a-i da semnificație. De exemplu, *<p>* definește un paragraf, *<a>* un link, iar ** o imagine. Acestea sunt în mod obișnuit organizate într-o structură ierarhică, pornind de la o etichetă principală *<html>*, care conține două secțiuni: *<head>* (cu metadata, titlu, legături către stiluri și scripturi) și *<body>* (unde se află efectiv conținutul afișat în pagină). [2]

HTML este strâns legat de CSS (pentru stilizare) și JavaScript (pentru interactivitate). Împreună, aceste trei tehnologii formează „scheletul” oricărei aplicații web. De-a lungul timpului, HTML a evoluat semnificativ, ajungând astăzi la versiunea HTML5, care introduce elemente semantice precum *<article>*, *<section>*, *<header>*, dar și suport nativ pentru video, audio, canvas și multe altele. [2]

2.7 Cascading Style Sheets (CSS)

CSS, prescurtare de la Cascading Style Sheets, este limbajul folosit pentru a stiliza și prezenta elementele HTML pe paginile web. A fost creat pentru a separa structura (HTML) de aspect (culori, fonturi, poziționare), permițând dezvoltatorilor să creeze interfețe coerente, atractive și responsive. [2]

CSS funcționează pe baza regulilor, fiecare formată dintr-un selector (care stabilește ce elemente HTML vor fi afectate) și un bloc de declarații (care definesc stilul, de exemplu: culoare, dimensiune, margini etc.). De exemplu, o regulă simplă care colorează tot textul paragrafelor în roșu ar arăta:

```
p {  
    color: red;  
}
```

Fiind „cascading” (în cascadă), CSS aplică regulile în funcție de mai mulți factori: specificitate, ordinea de apariție, și dacă stilurile sunt definite inline, în fișiere externe sau într-un `<style>` din HTML. [2]

CSS permite:

- controlul layout-ului paginii (prin *Flexbox*, *Grid* etc.);
- personalizarea completă a vizualului (fonturi, tranziții, animații);
- adaptarea conținutului pentru diferite dispozitive (prin media queries, pentru responsive design).

[2]

CSS poate fi organizat și modularizat cu ajutorul unor preprocesatoare (ex. SASS, LESS) sau poate fi gestionat direct în fișierele componentelor, cum se întâmplă în Blazor, React sau Angular. În Blazor, de exemplu, pot exista fișiere *.razor.css* care afectează stilul unei componente fără a influența alte părți ale aplicației. [2]

CSS este responsabil de aspectul și senzația vizuală a aplicației, completând structura oferită de HTML și funcționalitatea livrată de C#, JavaScript sau alte limbaje. [2]

2.8 Bootstrap

Bootstrap este un framework front-end open-source dezvoltat inițial de Twitter, lansat pentru prima dată în 2011, cu scopul de a oferi un set unitar și rapid de unelte pentru crearea interfețelor web moderne. Este construit în jurul a trei tehnologii principale: HTML, CSS și JavaScript, și oferă o colecție de componente predefinite, stiluri și funcționalități interactive care pot fi integrate rapid într-o aplicație. [2]

Unul dintre cele mai importante avantaje ale Bootstrap este sistemul său de grid responsive (bazat inițial pe 12 coloane), care permite construirea layout-urilor flexibile, adaptabile pentru orice dimensiune de ecran, de la telefoane mobile până la monitoare mari. [2]

Bootstrap include o varietate de:

- componente UI (buttoane, formulare, alerte, carduri, navigație etc.),
- plugin-uri JavaScript (dropdown-uri, carusele, modale, tooltip-uri etc.),
- variabile și mixin-uri Sass pentru personalizare avansată.

[2]

Dezvoltatorii pot folosi clase CSS predefinite (ex: *btn btn-primary*, *d-flex*, *text-center*) pentru a aplica rapid stiluri fără a scrie cod CSS de la zero. Această abordare reduce timpul de dezvoltare și asigură un aspect uniform al aplicației.[2]

În Blazor, Bootstrap este frecvent utilizat pentru a stiliza rapid componentele *.razor*. Fiind un framework agnostic (nu depinde de un anumit limbaj de programare back-end), poate fi integrat cu ușurință și în proiecte .NET, ASP.NET sau Blazor. În special în Blazor WebAssembly, este obișnuit să se includă Bootstrap direct în *index.html* și să se aplice clasele Bootstrap componentelor din *.razor*.

2.9 Blazor

Blazor este un framework open-source dezvoltat de Microsoft, lansat oficial în 2018, ca parte a ecosistemului .NET. Numele „Blazor” este format din cuvintele „Browser” și „Razor”, evidențiind utilizarea motorului Razor pentru redarea componentelor web direct în browser. [6]

Scopul principal al Blazor este de a permite dezvoltatorilor .NET să creeze aplicații web interactive fără a apela la JavaScript, oferind o alternativă la framework-uri front-end precum React, Angular sau Vue.js. Utilizând limbajul C# și întreg ecosistemul .NET, Blazor a devenit rapid o opțiune atractivă pentru dezvoltatorii familiarizați cu tehnologiile Microsoft, facilitând dezvoltarea de aplicații full-stack doar cu .NET. [6]

Blazor este disponibil în două variante principale:

- Blazor Server (2019): Aplicația rulează pe server, iar interacțiunea cu utilizatorul este gestionată în timp real prin SignalR. Această variantă oferă performanțe ridicate și un consum redus de resurse pe client, dar necesită o conexiune constantă la server.
- Blazor WebAssembly (2020): Codul C# este compilat în WebAssembly și rulează direct în browser, eliminând nevoia unei conexiuni continue la server. Aceasta permite dezvoltarea de aplicații web care pot funcționa și offline.

[6]

Utilizări principale ale Blazor:

- Aplicații web interactive (SPA - Single Page Applications): Blazor permite dezvoltarea de aplicații de tip SPA, în care navigarea și interacțiunile cu utilizatorul se realizează fără reîncărcarea completă a paginii, oferind o experiență fluidă și modernă.
- Aplicații WebAssembly: Cu Blazor WebAssembly, aplicațiile pot rula complet în browser, reducând latențele și oferind posibilitatea de a crea aplicații offline sau cu funcționare locală.
- Aplicații server-side: Blazor Server este ideal pentru aplicații care necesită control sporit asupra datelor și un răspuns în timp real. Prin SignalR, modificările din UI sunt reflectate instant, fără apeluri repetate la server.
- Aplicații enterprise: Datorită integrării excelente cu ecosistemul .NET, Blazor este preferat în mediul enterprise pentru reutilizarea codului existent, integrarea ușoară a logicii de business, autentificării, bazelor de date și serviciilor API.

[6]

În Blazor, HTML-ul este folosit într-un context puțin diferit: HTML-ul devine parte dintr-un fișier .razor, care combină markup-ul (HTML) cu C# într-o singură componentă.

Blazor este un framework dezvoltat de Microsoft, parte din ecosistemul .NET, care permite scrierea de aplicații web interactive folosind C# în loc de JavaScript. Cu toate acestea, interfața aplicației tot cu HTML se definește. De exemplu, pentru a crea un buton, un formular, o listă sau un container de conținut, se vor folosi aceleași etichete HTML standard, ca în orice aplicație web tradițională. [6]

Diferența esențială e că în Blazor:

- se pot lega date (*data binding*) direct în HTML folosind sintaxa @ (ex. @someProperty);
- se poate interacționa cu evenimente ca @onclick, @oninput, etc., legându-le direct la metode din C#;
- se pot include componente (echivalente cu funcții sau clase UI reutilizabile) folosind HTML-like tags: <MyComponent />.

[6]

De exemplu, în loc să se scrie un buton cu JavaScript care să modifice o valoare, în Blazor se va scrie un buton HTML cu un @onclick care apelează o metodă C#.

Proiectul de licență folosește Blazor WebAssembly pentru ca se dorește o platformă destul de light pentru utilizator care poate fi folosită și în zone scăzute de internet. Blazor Server folosește comunicarea cu un server comun prin SignalR ceea ce necesită o conexiune permanentă la internet. Iar la un volum mare de utilizatori, acest canal poate deveni lent.

2.10 MudBlazor

MudBlazor este o bibliotecă de componente UI open-source pentru Blazor, construită cu scopul de a oferi o experiență modernă și coerentă de design bazată pe Material Design, dar adaptată complet ecosistemului .NET. A fost creată pentru a permite dezvoltatorilor să construiască aplicații Blazor atractive și funcționale fără a apela la JavaScript extern sau framework-uri CSS precum Bootstrap. [17]

MudBlazor a fost gândit pentru Blazor și folosește avantajele acestuia, cum ar fi bindingul bidirecțional și componentizarea simplă, pentru a crea o experiență de dezvoltare fluidă. [17] Cu MudBlazor, se pot folosi componente precum:

- MudButton, MudTextField, MudSelect, MudDialog, MudTable, MudCard
- MudLayout și MudDrawer pentru crearea de interfețe responsive
- MudTheme pentru personalizare ușoară a temei aplicației (culori, fonturi, spacing)

[17]

Un mare avantaj este că toate componentele sunt complet compatibile cu codul C# și interacțiunile din Blazor, nu e nevoie de JavaScript pentru funcționalități precum dialoguri, notificări, meniuri sau validarea formularelor. [17]

MudBlazor vine și cu suport pentru:

- Dark mode nativ
- Validare fluentă a formularelor
- Tablouri de bord moderne cu grafică, animații și layout-uri fluide

[17]

Este ideal pentru aplicații enterprise sau proiecte în care se dorește o interfață consistentă și ușor de întreținut în timp. [17]

De asemenea, documentația MudBlazor este completă și oferă exemple clare pentru toate componentele.

2.11 SignalR

WebSockets este un protocol de comunicație care permite o conexiune bidirecțională, persistentă și full-duplex între un client (de exemplu, un browser web) și un server, facilitând transmiterea rapidă și continuă a datelor în timp real fără a reîncarca pagina. [12]

SignalR este o bibliotecă dezvoltată de Microsoft care facilitează comunicarea în timp real între aplicațiile web, mobile sau desktop și servere. Lansată în 2011 și integrată ulterior în

ecosistemul ASP.NET Core, SignalR permite actualizări și notificări instantanee fără a fi nevoie de reîncărcarea paginile web. [12]

Înainte de SignalR, comunicarea bidirecțională în timp real era dificilă și adesea implementată prin tehnici ineficiente precum polling sau long-polling, care consumau multe resurse. SignalR a simplificat acest proces prin integrarea automată a protocolului WebSockets, care oferă o conexiune persistentă și eficientă între client și server. [12]

Cu apariția ASP.NET Core, SignalR a fost reproiectat pentru a fi mai performant și scalabil, suportând diverse metode de transport și oferind o experiență optimă indiferent de mediu. [12]

SignalR folosește automat cel mai potrivit mecanism de comunicare, în funcție de capabilitățile clientului și ale serverului:

- WebSockets: Protocolul principal, oferind o conexiune rapidă și bidirecțională.
- Server-Sent Events (SSE): Permite serverului să trimită actualizări către client printr-o conexiune HTTP deschisă.
- Long Polling: Metoda de rezervă, în care clientul face cereri repetate pentru a verifica noutățile atunci când celelalte opțiuni nu sunt disponibile.

[12]

SignalR este ideal pentru aplicații care necesită actualizări în timp real, cum ar fi chat-uri, notificări, dashboard-uri live sau colaborare online. Acesta ajută componentele proiectului FastRide să comunice fără a fi nevoie de acțiuni manuale ale utilizatorului sau reîncărcarea paginii.

2.11.1 SignalR Server Emulator

SignalR este o bibliotecă dezvoltată de Microsoft care facilitează comunicarea bidirecțională în timp real între server și client. Este folosită în aplicații precum chat-uri, sisteme de notificare live, jocuri multiplayer sau aplicații colaborative. [13]

În timpul dezvoltării, conectarea la un serviciu SignalR real (precum Azure SignalR Service) poate fi:

- mai costisitoare (în special pentru testare frecventă),
- dependentă de internet sau de configurări complexe,
- dificil de controlat în ceea ce privește scenariile de test (ex. simularea mai multor utilizatori sau pierderi de conexiune).

[13]

SignalR Server Emulator este un instrument util în special în etapa de dezvoltare și testare a aplicațiilor web sau mobile care folosesc comunicarea în timp real prin SignalR.

Scopul său este să emuleze comportamentul unui server SignalR real, astfel încât dezvoltatorii să poată testa interacțiuni în timp real (mesaje, notificări, actualizări de stare etc.) fără a depinde de un back-end implementat complet sau de infrastructura Azure. [13]

SignalR Server Emulator rezolvă probleme enumerate prin rularea locală, oferirea unei interfețe sau API-uri care simulează semnalarea și transmiterea mesajelor, integrarea ușoară cu aplicațiile existente care folosesc SignalR. [13]

Emulatorul pornește un hub SignalR local (de obicei într-un ASP.NET Core server simplificat) care acceptă conexiuni WebSocket de la clienți (browser sau aplicații mobile), trimite mesaje simulate (manual sau automat), permite testarea scenariilor cum ar fi broadcast, mesaje către grupuri, reconectări sau deconectări. [13]

Totuși emulatorul nu înlocuiește complet un serviciu real precum Azure SignalR. Nu oferă scalabilitate mare, autentificare complexă, routing distribuit sau integrare cu alte servicii cloud. [13]

Pentru a putea face posibilă testarea aplicației prezente cu o conexiune SignalR, atunci se folosește SignalR Emulator.

2.12 Refit API

Refit (abreviere de la Rest Service Interface) este o bibliotecă open-source pentru platforma .NET, dezvoltată cu scopul de a simplifica interacțiunea aplicațiilor cu API-uri REST. Aceasta permite definirea interfețelor API într-un mod declarativ, bazat pe atribute, înlocuind codul tradițional de apel HTTP scris manual cu o abordare mult mai concisă și lizibilă. [18]

Refit a fost inspirat de biblioteca Retrofit, utilizată în ecosistemul Android. A fost creată de Paul Betts și continuă să fie întreținută de comunitatea .NET. Scopul său este acela de a automatiza procesul de trimitere a cererilor HTTP și de a deserializa răspunsurile în obiecte C#, oferind astfel o metodă elegantă și productivă de a lucra cu API-uri externe. [18]

Refit se bazează pe conceptul de interfață C# decorată cu atribute, care definește metodele corespunzătoare endpoint-urilor unui serviciu REST. La momentul execuției, biblioteca generează automat o implementare a acestei interfețe folosind *RestService.For<T>()*. Această implementare utilizează intern *HttpClient*, dar ascunde complet complexitatea acestuia. [18]

2.13 Leaflet și Routing Machine

Leaflet este o bibliotecă JavaScript open-source pentru hărți interactive, recunoscută pentru dimensiunea redusă, performanța ridicată și ușurința în utilizare. A fost creată pentru a fi ușor de integrat în aplicații web și este folosită pe scară largă în proiecte care necesită afișarea hărților și a datelor geospațiale într-un mod interactiv. Datorită versatilității sale, Leaflet poate funcționa eficient pe toate browserele moderne și oferă suport nativ pentru interacțiuni uzuale

precum zoom, pan, adăugarea de markere și pop-up-uri. [5]

Pentru aplicațiile care necesită trasarea unei rute între două sau mai multe puncte, se poate integra pluginul Leaflet Routing Machine, care adaugă funcționalitatea de navigare și calcul de rute direct în hartă. Acesta se bazează în general pe servicii externe precum OSRM (Open Source Routing Machine), GraphHopper sau Mapbox Directions pentru a calcula rutele, oferind o funcționalitate ce include afișarea traseelor, inclusiv cu indicații pas cu pas și actualizări în timp real. [5]

Într-o aplicație web, Leaflet și Routing Machine pot lucra împreună pentru a permite utilizatorilor să vizualizeze pe hartă locații relevante, să genereze rute între puncte selectate și să primească detalii despre distanță, durată și drumuri parcurse și totul gratis, de aceea se utilizează și în proiectul FastRide.

2.14 Stripe

Stripe este o platformă globală de procesare a plăților, fondată în 2010 de frații Patrick și John Collison. Aceasta a fost creată pentru a permite afacerilor și dezvoltatorilor să accepte plăți online în mod simplu și sigur. Stripe a devenit rapid unul dintre cei mai populari furnizori de soluții de plăți digitale, datorită ușurinței de integrare, suportului pentru diverse metode de plată și disponibilității în multiple țări. [19]

Stripe a fost fondată într-o perioadă în care comerțul online era în creștere, dar soluțiile de plată disponibile erau adesea greoaie sau complexe de implementat. Frații Collison au observat o oportunitate de a crea o platformă care să facă procesul de integrare a plăților în site-uri web și aplicații mult mai simplu pentru dezvoltatori. Stripe a fost lansată oficial în 2011, cu misiunea de a moderniza plățile online și de a oferi soluții intuitive pentru afaceri de toate dimensiunile. [19]

Stripe este o platformă populară pentru procesarea plăților online, iar în cod, interacțiunea cu Stripe se face de obicei prin intermediul unui SDK oficial (cum ar fi Stripe.net în C#), care comunică cu API-ul Stripe. Tot fluxul pornește de la inițierea unei acțiuni de plată sau de verificare a cardului, iar implementarea în cod respectă pașii tipici ai unui flux de plată securizat și asincron. [19]

Într-un flux obișnuit, aplicația back-end creează un *PaymentIntent* sau un *SetupIntent*, în funcție de ceea ce se dorește, adică dacă se vrea să se încaseze o sumă sau doar să se verifice și să se salveze un card pentru o plată ulterioară. Odată ce intent-ul a fost creat, back-end-ul primește un *client_secret*, care este trimis către front-end. [19]

Pe front-end, Stripe.js sau Stripe Elements este folosit pentru a colecta datele cardului și a le trimite către Stripe, fără ca serverul să vadă sau să stocheze datele sensibile. Cu acel *client_secret*, front-end-ul finalizează interacțiunea și Stripe se ocupă de autorizarea plății sau a cardului. [19]

După ce Stripe procesează totul, trimite un răspuns către back-end prin webhook-uri (de

exemplu, când o plată a fost finalizată cu succes, sau când un card a fost salvat), iar aplicația poate reacționa la aceste evenimente, de exemplu, salvând un ID de client Stripe sau marcând o comandă ca fiind plătită. [19]

Codul doar configurează cererea, transmite comenzile către Stripe și reacționează la rezultatele oferite de Stripe prin webhook-uri. Acest model oferă securitate, scalabilitate și conformitate cu standardele precum PCI-DSS.

3. ELEMENTE SOFTWARE FOLOSITE

3.1 JetBrains Rider IDE

JetBrains Rider este un mediu de dezvoltare integrat (IDE) dezvoltat de compania JetBrains, binecunoscută pentru IntelliJ IDEA și alte instrumente populare dedicate dezvoltatorilor. Rider a fost lansat oficial în 2017, având ca obiectiv principal să ofere o alternativă performantă și multiplatformă pentru dezvoltarea aplicațiilor .NET, compatibilă cu Windows, macOS și Linux. [4]

Rider combină motorul de analiză a codului de la ReSharper (foarte cunoscut în rândul utilizatorilor de Visual Studio) cu platforma IntelliJ, rezultând astfel un IDE rapid, stabil și bogat în funcționalități. Este folosit pe scară largă în dezvoltarea de aplicații .NET Core, ASP.NET, Xamarin, Unity și Blazor. [4]

Una dintre trăsăturile importante ale JetBrains Rider este faptul că este un IDE orientat pe comenzi rapide de la tastatură (keyboard-centric). Aproape orice acțiune poate fi executată rapid fără a naviga prin meniuri, iar comenzile implicite pot fi personalizate. Rider oferă o varietate de *keymaps* predefinite (configurații de comenzi rapide), pentru a se potrivi stilului fiecărui dezvoltator. [4]

Interfața principală a aplicației este simplă și curată, dar oferă acces rapid la cele mai importante funcționalități: construirea soluției, configurarea modului de rulare/debug, integrarea cu sisteme de control al versiunilor (Git, SVN), precum și funcționalități de căutare globală sau localizare a fișierelor. [4]

Rider este capabil să lucreze nativ cu tehnologii precum Docker, baze de date SQL, debug remote, testare unitară și CI/CD. Este compatibil cu majoritatea toolurilor moderne folosite în dezvoltare și o viteză de analiză a proiectelor mari ridicată. [4]

În proiectul FastRide, JetBrains Rider a fost utilizat ca IDE principal pentru dezvoltarea back-end-ului scris în C#, gestionarea componentelor Azure Functions și stilizarea UI-ului.

3.2 Microsoft Azure Storage Explorer

Azure Storage Explorer este un instrument grafic, dezvoltat de Microsoft, care permite accesul și gestionarea datelor stocate în Azure Storage, fie că este vorba despre conturi de stocare, containere, fișiere, tabele sau cozi de mesaje. Lansat pentru a simplifica interacțiunea cu serviciile de stocare din Azure, Azure Storage Explorer oferă o interfață ușor de utilizat pentru dezvoltatori și administratori, permițându-le să gestioneze eficient datele și resursele de stocare în cloud. [7]

Azure Storage Explorer a fost lansat de Microsoft în 2015, ca un instrument destinat

dezvoltatorilor și profesioniștilor IT care utilizează serviciile de stocare oferite de Azure. Înainte de apariția acestuia, interacțiunea cu Azure Storage se realiza în principal prin intermediul Azure Portal, folosind interfețe web sau scripturi și API-uri. Pentru a oferi o soluție mai intuitivă și accesibilă pentru gestionarea resurselor de stocare, Microsoft a dezvoltat Azure Storage Explorer, un client desktop disponibil pentru Windows, macOS și Linux. [7]

Acesta a evoluat de-a lungul timpului, integrându-se cu alte servicii din Azure și oferind funcționalități extinse, cum ar fi gestionarea datelor offline, suport pentru acces la mai multe conturi de stocare și securizarea accesului prin autentificare bazată pe Azure Active Directory AAD. [7]

Pentru proiectul prezent, această aplicație facilitează navigarea developer-ului prin baza de date a lucrării.

3.3 Docker

Docker Desktop este o aplicație ușor de instalat, disponibilă pentru Windows, macOS și Linux, care permite dezvoltatorilor să creeze, să partajeze și să ruleze aplicații containerizate sau microservicii foarte ușor. [1]

Unul dintre principalele avantaje ale Docker Desktop este interfața grafică simplă și intuitivă (GUI), care face ca gestionarea containerelor, a aplicațiilor și a imaginilor să fie accesibilă chiar și pentru cei fără experiență avansată în administrarea de medii virtualizate. Astfel, dezvoltatorii pot urmări ce rulează local, pot porni/opri containere sau pot inspecta fișierele și logurile cu ușurință, direct din interfață. [1]

Prin utilizarea Docker Desktop, se elimină o mare parte din timpul pierdut pe configurări complexe. Aplicația gestionează automat detalii precum porturile, sistemul de fișiere, rețelele interne și alte setări implicite, oferind un mediu de dezvoltare stabil și actualizat constant cu patch-uri de securitate și corecturi de erori. [1]

Un alt avantaj major este integrarea nativă cu majoritatea limbajelor și uneltelor de dezvoltare moderne. În plus, Docker oferă acces la Docker Hub, o platformă online cu imagini oficiale și comunitare, pe care dezvoltatorii o pot folosi pentru a construi rapid prototipuri, a automatiza procesul de build sau a implementa fluxuri de CI/CD (Continuous Integration / Continuous Deployment). [1]

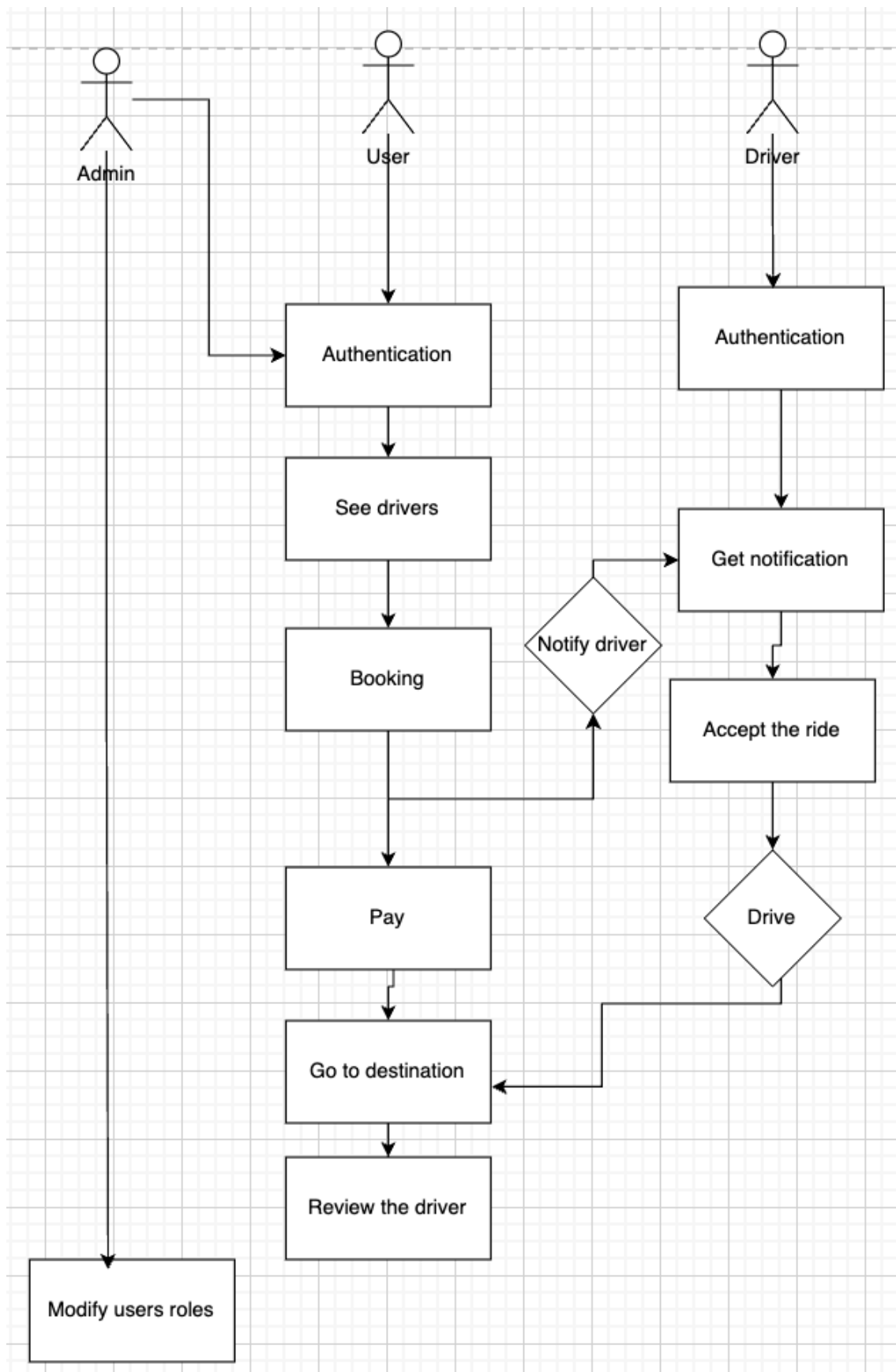
În proiectul FastRide, Docker a fost folosit pentru a hosta storage-ul aplicației (Azurite).

4. FLUXUL APLICAȚIEI

Aplicația FastRide este concepută pentru a facilita rezervarea și acceptarea curselor între clienți și șoferi într-un mod rapid, sigur și intuitiv. Utilizatorul se autentifică cu un cont Google, își poate introduce un număr de telefon (dacă nu a fost deja preluat automat) și poate începe imediat să interacționeze cu platforma. Clienții pot vizualiza harta, alege punctul de plecare și destinația, iar aplicația va calcula ruta și va căuta șoferi disponibili în apropiere. Odată ce un șofer acceptă cursa, aceasta devine activă și poate fi urmărită în timp real. La final, clientul poate oferi un rating, iar istoricul curselor este salvat pentru consultare ulterioară.

Diagrama *Use Case* reflectă interacțiunile principale dintre utilizatori și sistem. Există trei tipuri de actori: clientul (*User*), șoferul (*Driver*) și administratorul (*Admin*). Clientul poate efectua acțiuni precum: autentificare, rezervare cursă, vizualizare rută, urmărire în timp real și acordare de rating. Șoferul, la rândul său, se autentifică, primește notificări cu cereri de curse, le poate accepta, vede traseul și finalizează cursa. Iar administratorul, în plus, poate să asigneze roluri utilizatorilor. Toate aceste interacțiuni sunt coordonate de sistemul back-end, care se ocupă de procesarea cererilor, actualizarea în timp real a stărilor și salvarea datelor în storage.

În analiza cazurilor de utilizare, diagrama *Use Case* nu doar conturează funcționalitățile sistemului, ci evidențiază și relațiile de dependență între acțiuni și actori. Un aspect important vizibil în diagramă este separarea clară a responsabilităților, ceea ce permite o arhitectură ușor de întreținut. De exemplu, legătura dintre client și acțiunea de evaluare a șoferului apare doar după încheierea unei curse, ceea ce indică un flux condiționat logic. Similar, interacțiunea șoferului cu cursa, este posibilă doar în momentul în care aceasta a fost creată și nu este deja acceptată, ceea ce reflectă restricții de business transpuse clar în comportamentul aplicației. În acest fel, diagrama devine nu doar o hartă a funcționalităților, ci și o expresie a regulilor din spatele aplicației.

**Fig. 4.1.** Diagrama *Use case* a proiectului.

Interfața este una prietenoasă oferind utilizatorului toate informațiile de care are nevoie. Aplicația este *responsive* și este suportată atât pe desktop cât și pe mobile.

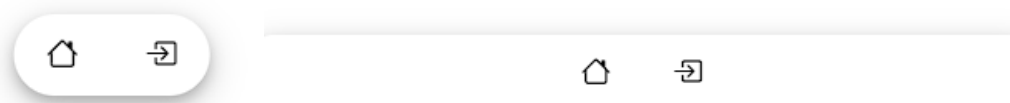


Fig. 4.2. Bara de navigare pe desktop și mobile.

4.1 Experiența utilizatorului

Primul pas pe care toți utilizatorii trebuie să îl facă pentru a beneficia de funcționalitățile proiectului, este să se autentifice.

Utilizatorul se autentifică în aplicație, prin contul personal Google, apăsând butonul de *login* din bara de navigare. Fără autentificare, utilizatorul poate doar să vadă locația curentă și să caute diverse locații pe hartă.



Fig. 4.3. Mark-ul *human* ce indică locația actuală a utilizatorului.

După autentificare, utilizatorul își poate rezerva o cursă. Acest lucru se realizează prin plasarea unui *pin* pe hartă, ce indică locul unde dorește să fie lăsat.



Fig. 4.4. Mark-ul *pin* pentru a sugera destinația.

Pentru a realiza acest lucru, fie se face click pe hartă, fie se folosește bara de search cu sugestii.

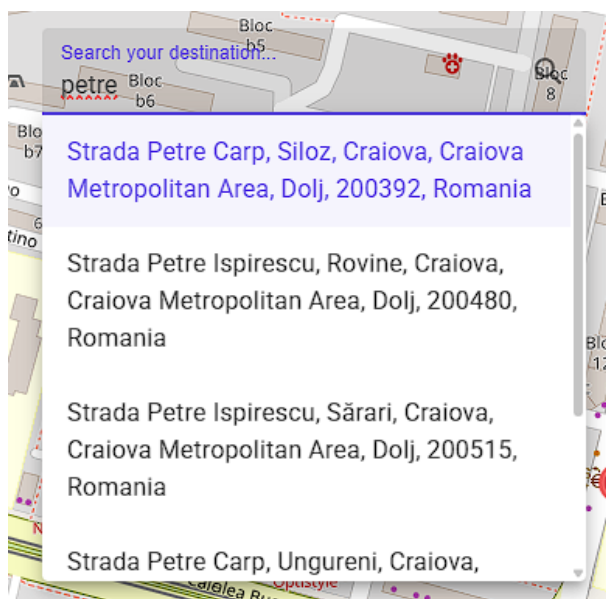


Fig. 4.5. Bara de căutare pentru adrese cu sugestii.

Utilizatorul poate apăsa butonul *Ride* pentru a porni cursa. Pe hartă se afisează și ceilalți șoferi ce sunt disponibili pentru a accepta o cursă din aceeași localitate.



Fig. 4.6. Mark-ul *driver* ce semnifică un utilizator de tip șofer disponibil.

Odată apăsat butonul, utilizatorul trebuie să treacă prin doi pași: să confirme cursa și să ofere informații despre cardul bancar. După apăsarea butonului și pregătirea cursei în server, un pop-up cu toți pașii i se deschide și trebuie completat. Dacă ceva nu este în ordine, utilizatorul primește feedback în legătura cu erorile.

The ride costs: **LEI 3.62**
Do you want to continue?

× →

● ————— 🚗

Card Google Pay

Card number
4242 4242 4242 4242 **VISA**

Expiration date (MM/YY) Security code
11 / 21 111 123

Your card's expiration year is in the past.

Country
Romania ▼

× →

Fig. 4.7. Pașii pe care utilizatorul trebuie să îi completeze.



Payment is completed! Search a rider is in progress...

OK

Fig. 4.8. Mesajul de confirmare că totul este în regulă.

Mai departe, utilizatorul așteaptă pentru ca un șofer să îi accepte cursa. În caz că nu

se găsește un șofer atunci cursa este anulată. Fiecare status de anulare se observă în meniul de informare, ce ține locul bării de căutare.

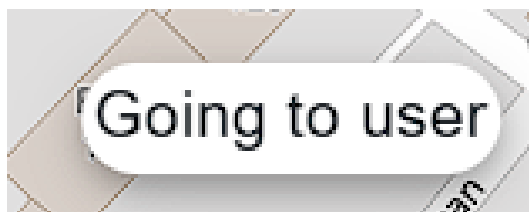


Fig. 4.9. Meniul de informare al status-ului.

Dacă s-a găsit un șofer, atunci utilizatorului i se deschide un pop-up cu detaliile despre acesta și apoi este nevoie să aștepte ca șoferul să ajungă să îl preia. După ce clientul se află în mașina șoferului, pot să meargă împreună spre destinație.

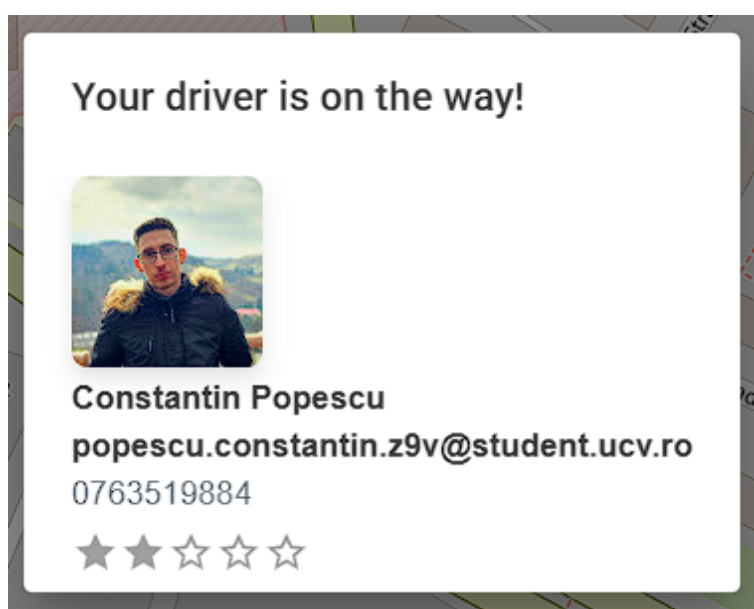


Fig. 4.10. Detaliile despre șofer după ce acceptă cursa.

Utilizatorul poate vedea ruta pe unde șoferul o să meargă, iar *pin*-ul ce indică locația curentă se modifică în *currentCar*.



Fig. 4.11. Mark-ul *currentCar* ce semnifică că utilizatorul călătorește cu mașina.

La final de cursă, opțional, clientul poate lăsa o recenzie șoferului.

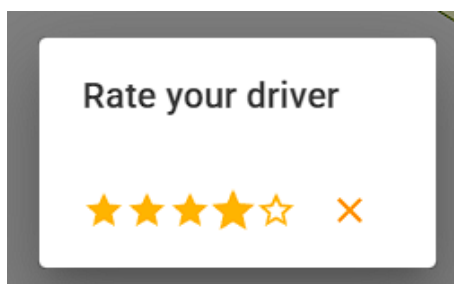


Fig. 4.12. Dialog-ul ce permite oferirea unui recenzii șoferului.

Alte lucruri pe care le poate face clientul, dar și celelalte tipuri de utilizatori, este să își schimbe numărul de telefon sau să navigheze către Google Accounts pentru a își schimba numele sau imaginea de profil.

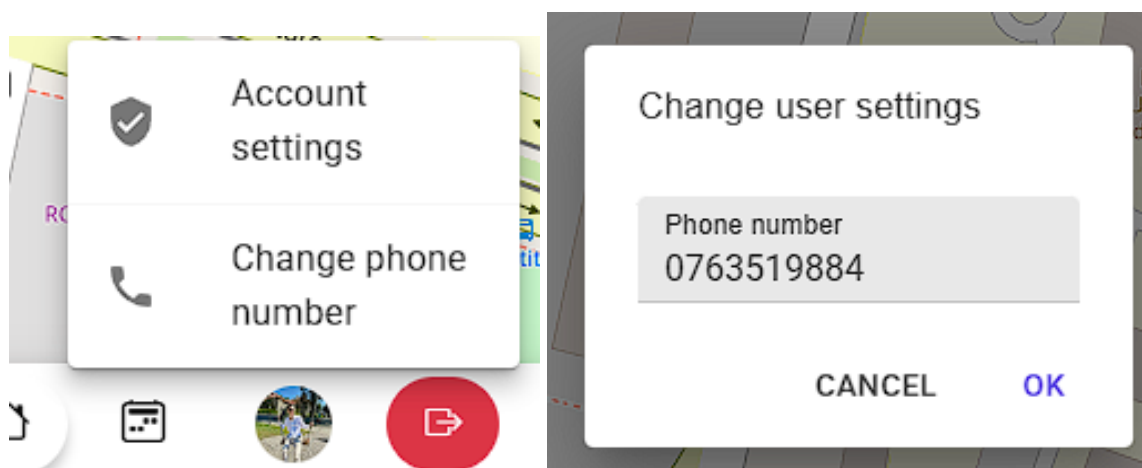


Fig. 4.13. Schimbarea numărului de telefon apăsând pe imaginea de profil.

Utilizatorii pot vedea și istoricul curselor prin apăsarea butonului din bara de navigare. Pentru fiecare cursă se poate vedea adresa de destinație, data și ora la care s-a completat cursa, prețul plătit și, în cazul de anulare a cursei, motivul anulării. Folosind butonul *Rebook*, utilizatorul poate să repună *pin*-ul în același loc precum în istoric.

My rides

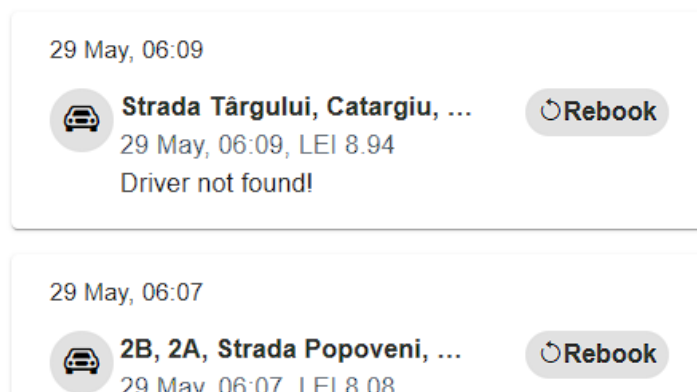


Fig. 4.14. Istoricul curselor.

4.2 Experiența șoferului

Șoferul nu are accesul să rezerve o cursă, însă este asignat să accepte una, de aceea flow-ul acestuia începe după ce un client începe o cursă și îi se atribuie cererea. Bineînțeles, trebuie să fie autentificat.

El are un alt buton, *GetRides* care deschide un dialog unde poate vedea cursele asignate ce i sunt asignate.

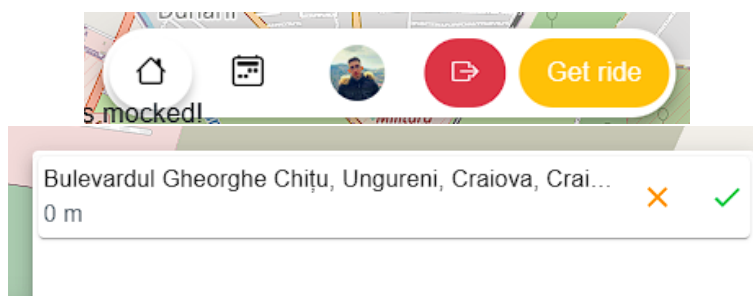


Fig. 4.15. Bara de navigare împreună cu dialogul-ul ce afișează ofertele curente.

Singura responsabilitate a șoferului, după ce a acceptat cursa, este să meargă la client, să îl aștepte și să meargă împreună spre destinație. În acești pași, șoferul are prima destinație către client, apoi către destinația propriu zisă.

Ca și ajutor în trafic, acesta poate să vadă ruta spre punctul de destinație, actualizându-se în timp real și poate regăsi și indicații de navigare.

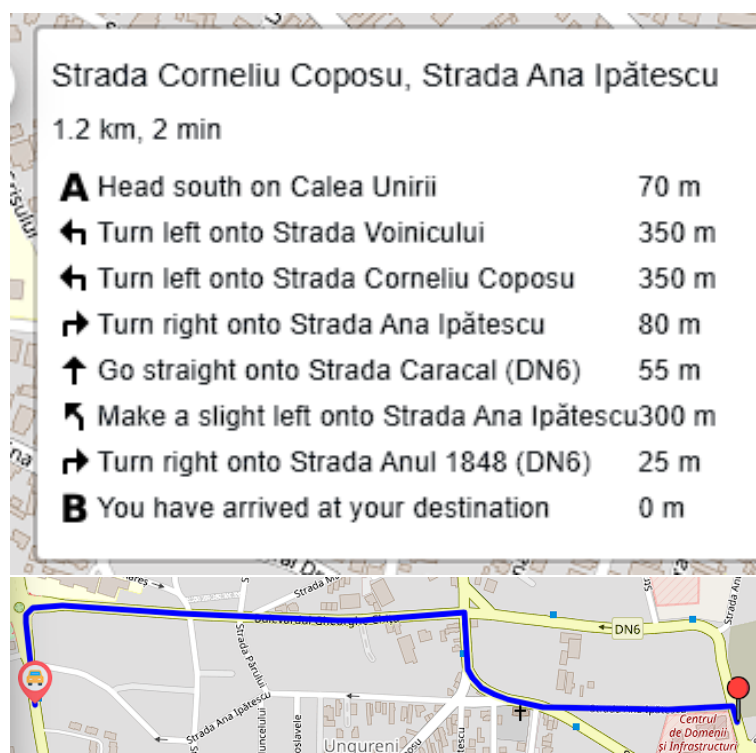


Fig. 4.16. Exemplu de rută și indicații de navigare.

4.3 Experiența admin-ului

În plus de cele menționate la utilizatorul normal, administratorul poate schimba tipul oricărui utilizator, iar, de asemenea, mai poate actualiza și numărul de telefon. Prin apăsarea butonului din bara de navigare, i se deschide un tabel unde poate vedea toate informațiile despre utilizatori. El poate să-și filtreze utilizatorii după numărul de telefon, nume, email sau identificator.

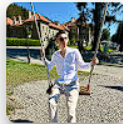
Users						
Q mada						
Identifier ↑	Email	User Name	Phone Number	Role	Rating	Picture
d6088988-fe33-fd31-87b8-d42198b8d245	dizzyforester@gmail.com	Madalin Popescu	076351988	User	0	
				Driver		
Rows per page: 10 1-1 of 1						

Fig. 4.17. Pagina administratorului.

Aplicația FastRide oferă o experiență intuitivă și eficientă pentru utilizatorii care doresc să rezerve sau să accepte curse într-un sistem de ride-sharing. Printr-un proces simplificat de

autentificare cu contul Google, utilizatorii pot accesa rapid funcționalitățile esențiale, precum vizualizarea hărții, stabilirea destinației, comunicarea în timp real și evaluarea curselor. Interfața este ușor de folosit, iar integrarea serviciilor externe precum Stripe, SignalR și Leaflet contribuie la o funcționare fluidă și sigură, atât pentru clienți, cât și pentru șoferi.

5. ARHITECTURA APLICAȚIEI

Aplicația FastRide este construită folosind o arhitectură scalabilă, bazată pe microservicii și servicii în cloud. Tehnologiile principale utilizate sunt Blazor WebAssembly pentru interfața utilizator și Azure Durable Functions pentru logica de back-end. Comunicarea dintre cele două componente este realizată prin API-uri HTTP și prin WebSocket-uri cu ajutorul SignalR.

Aplicația dispune de trei componente:

- stocare - unde se păstrează informațiile;
- server-side - unde se prelucrează toate informațiile;
- client-side - unde se afișează toate informațiile către utilizatori.

5.1 Comunicarea între componente

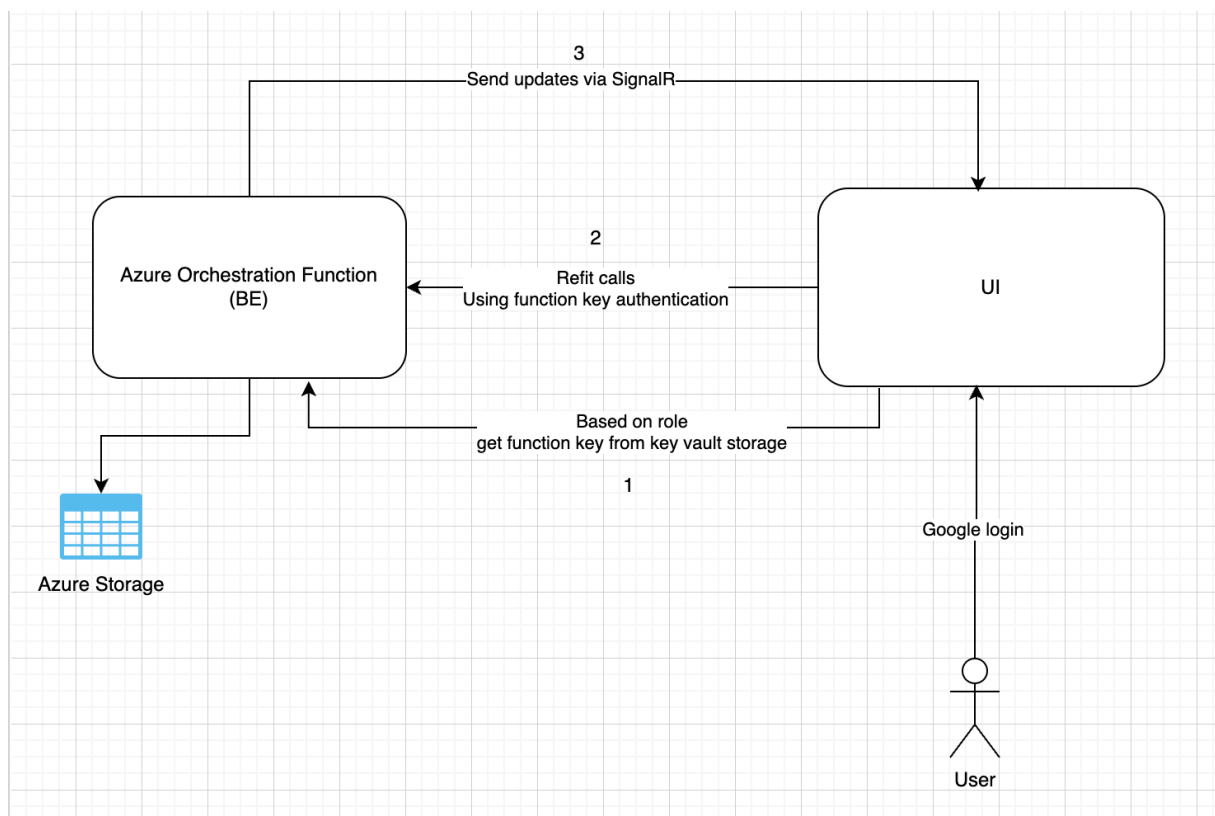


Fig. 5.1. Comunicarea dintre componentele proiectului.

Componentele proiectului comunică în perechi de câte două: Server-ul fiind cel ce gestionează atât Client-ul cât și Storage-ul.

Comunicarea cu Client-ul se face prin două căi: HTTP request și SignalR.

Comunicarea prin request-uri HTTP, se realizează prin librăria *Refit* ce permite serializarea și deserializarea request-ului și response-ului într-un mod mult mai ușor de folosit, apelarea și autorizarea endpoint-urilor de pe back-end fără a fi nevoie de crearea unui *HttpClient* și configurarea acestuia. Librăria doar face public un contract astfel încât, celelalte aplicații pot folosi doar ce li se dispune. Pentru acest lucru, s-a creat un proiect comun între cele două componente, numit *SDK*. Acesta poate fi folosit și ca *NuGet Package* pentru separarea soluțiilor în repository-uri diferite.

Proiectul se folosește de o clasă abstractă intermediară și un client ce customizează interfața *Refit*, pentru a reuși să aducă toate response-urile într-un anumit format (*ApiResponseMessage*).

```
12  public async Task<ApiResponseMessage<T>> Execute<T>(Task<T> task)
13  {
14      try
15      {
16          T response = await task;
17          OnApiCallExecuted(new ApiResponseMessage(success: true));
18          return new ApiResponseMessage<T>(success: true, response);
19      }
20      catch (ApiException ex)
21      {
22          OnApiCallExecuted(new ApiResponseMessage(success: false, ex.Status));
23          return new ApiResponseMessage<T>(success: false, response: Activat
24              responseMessage: ex.ReasonPhrase + " ; " + ex.Content);
25      }
26      catch (Exception ex)
27      {
28      }
29  }
30  var task = _apiClient.GetCurrentUserAsync();
31  var result :ApiResponseMessage<User> = await Execute(task);
```

Fig. 5.2. Metoda ce obține rezultatul dorit în urma execuției endpoint-ului.

Refit are nevoie doar de o interfață pentru a reuși să inițieze request-ul, unde trebuie să se specifice ruta, tipul request-ului și eventuali parametri împreună cu tipul răspunsului.

```

9  ✓ | public interface IFastRideApi
10  | {
11  |     [Get( path: "/api/user")]
12  |     Task<User> GetCurrentUserAsync();
13  |
14  |     [Post( path: "/api/user")]
15  |     Task<User> GetUserAsync([Body] UserIdentifier userIdentifier);

```

Fig. 5.3. Interfața Refit.

Pentru a autoriza orice request, Client-ul trebuie să ofere o implemetare pentru un *DelegatingHandler*, ce are rolul să interferează, ca un interceptor, la orice request făcut din interiorul host-ului și să adauge pe request *Authorization headers*.

```

17  |     services.AddRefitClient<IFastRideApi>()
18  |         .ConfigureHttpClient(c => c.BaseAddress = url)
19  |         .AddHttpClient<THttpDelegatingHandler>();
38  |     builder.Services.AddFastRideApiClient<HttpAuthenticationHandler>(
39  |         new Uri(builder.Configuration["FastRide:BaseUrl"]!));
53  |     request.Headers.Add( name: "Authentication", $"Bearer {tokenId}");
54  |     request.Headers.Add( name: "Authorization", $"Bearer {accessToken.Value}");

```

Fig. 5.4. Înregistrarea DelegatingHandler-ului din Client (*THttpDelegatingHandler* este *HttpAuthenticationHandler*) și adăugarea header-urilor de autentificare și autorizare.

Comunicarea prin SignalR se realizează fără un intermediar. Server-ul tratează mesajele de intrare și de ieșire folosind *Functions*, așa cum s-a arătat în descrierea componentei Server.

Pe partea de client, toate aceste mesaje se tratează folosind un serviciu. Astfel, se instanțiază o conexiune cu server-ul SignalR și se crează subscripții pentru mesajele de pe Server. Fiecare subscripție notifică un *event* din serviciu, care este, mai departe, ascultat de componente pentru a putea actualiza interfața.

```

63      _connection = new HubConnectionBuilder()
64          .WithUrl($"{_configuration["FastRide:BaseUrl"]}!}/api/?userId={userId}",
65          {
66              if (_configuration["FastRide:BaseUrl"]!.Contains("ngrok-free.app"))
67              {
68                  options.Headers.Add("ngrok-skip-browser-warning", "true");
69              }
70          })
71          .WithAutomaticReconnect() // IHubConnectionBuilder
72          .Build(); // HubConnection

85      _connection.On<NotifyUserGeolocation>( methodName: SignalRConstants.ServerNotifyUserGeolocation,
86          async (payload :NotifyUserGeolocation ) =>
87          {
88              if (NotifyDriverGeolocation != null!)
89              {
90                  await NotifyDriverGeolocation(payload);
91              }
92          });

```

Fig. 5.5. Conectarea la server-ul SignalR și crearea subscripției pentru notificarea noii locații a unui șofer pe hartă.

Pentru a trimite mesaje către Server (adică un alt utilizator) trebuie doar să se specifice numele evenimentului și să se pună în ordine parametrii pe care îi așteaptă Server-ul.

```

188      public async Task NotifyDriverArrivedAsync(string groupName)
189      {
190          await _connection.SendAsync( methodName: SignalRConstants.ClientDriverArrived, groupName);
191      }

```

Fig. 5.6. Trimiterea notificării către utilizator că s-a ajuns la unul din checkpoint-uri.

Pentru a ști ce tipuri de evenimente se folosesc, în proiectul de contracte pentru Refit s-a adăugat și o clasă de constante ce conține toate evenimentele existente în aplicație. Acestea au o format comun: *{source}.{action}*. Spre exemplu *client.join-user-group* este compus din sursa *client* ce semnifică faptul că mesajul este trimis de pe Client pe Server și acțiunea *join-user-group* ce reprezintă cererea de a înscrie un utilizator în grupul ce se trimite ca parametru.

Comunicarea dintre Server și storage se face folosind *Repository pattern*. Implementările conțin un client ce comunică cu Azure Storage. Acest client este o interfață *ITableClient<TEntity>* ce suportă metode de GET, ADD/UPDATE și DELETE. Pentru acestea, se folosește *TableClient* din Azure ce face HTTP request-uri către Azure Storage pentru a îndeplini acțiunile dorite.

Pentru ca Azure să recunoască modelele ce reprezintă entități, acestea trebuie să implementeze interfața *ITableEntity* ce conține cheile primare (*PartitionKey* și *RowKey*).

Pentru a oferi un nume tabelului, s-a creat un atribut ce poate fi folosit pe toate entitățile:

```

5      [AttributeUsage( validOn: AttributeTargets.Class | AttributeTargets.Property)]
6      ^o public class TableNameAttribute : Attribute
7      {
8          private string _name;
9
10         [2 usages  DizzYForester]
11         public TableNameAttribute(string name)
12         {
13             _name = name;
14         }
15
16         [1 usage  DizzYForester]
17         public string Name => _name;
18     }
19     DizzYForester, 10.11.2024, 00:51 • startup

```

Fig. 5.7. Atributul ce atribuie entității un nume de tabel pentru storage

```

10      [TableName(TableNames.Users)]
11      ^, public class UserEntity : ITableEntity
12      {
13          /// <summary>
14          /// Gets or sets the email.
15          /// </summary>
16          [7 usages]
17          public string PartitionKey { get; set; }
18
19          /// <summary>
20          /// Gets or sets the name identifier.
21          /// </summary>
22          [12 usages]
23          public string RowKey { get; set; }

```

Fig. 5.8. Cum se folosește atributul *TableName* pe entitatea *UserEntity*.

Comunicarea dintre componentele aplicației FastRide este esențială pentru buna funcționare a întregului sistem. Fiecare acțiune a utilizatorului declanșează un flux bine definit, în care front-end-ul, back-end-ul și serviciile externe colaborează pentru a livra o experiență coerentă și rapidă. În continuare, se descrie modul în care aceste componente interacționează între ele, punând accent pe orchestrarea proceselor și transmiterea în timp real a informațiilor.

5.2 Stocare

Pentru a memora toate datele și informațiile, *storage*-ul este esențial și una din componentele cele mai importante ale aplicației. Acesta este reprezentat de arhitectura Azure, folosind *Azure Storage* pentru rapiditate și costuri minime.

Azure Table Storage este o bază de date NoSQL, ideală pentru scenarii în care se lucrează cu volume mari de date structurate, dar fără relații complexe între entități. În cazul aplicației FastRide, fiecare entitate: utilizatori, curse, șoferi online și instanțele de orchestrare, este reprezentată ca o tabelă separată în Table Storage. Fiecare înregistrare (sau entitate) dintr-o tabelă este identificată în mod unic printr-o combinație de *PartitionKey* și *RowKey*, ceea ce asigură o distribuție eficientă și acces rapid la date.

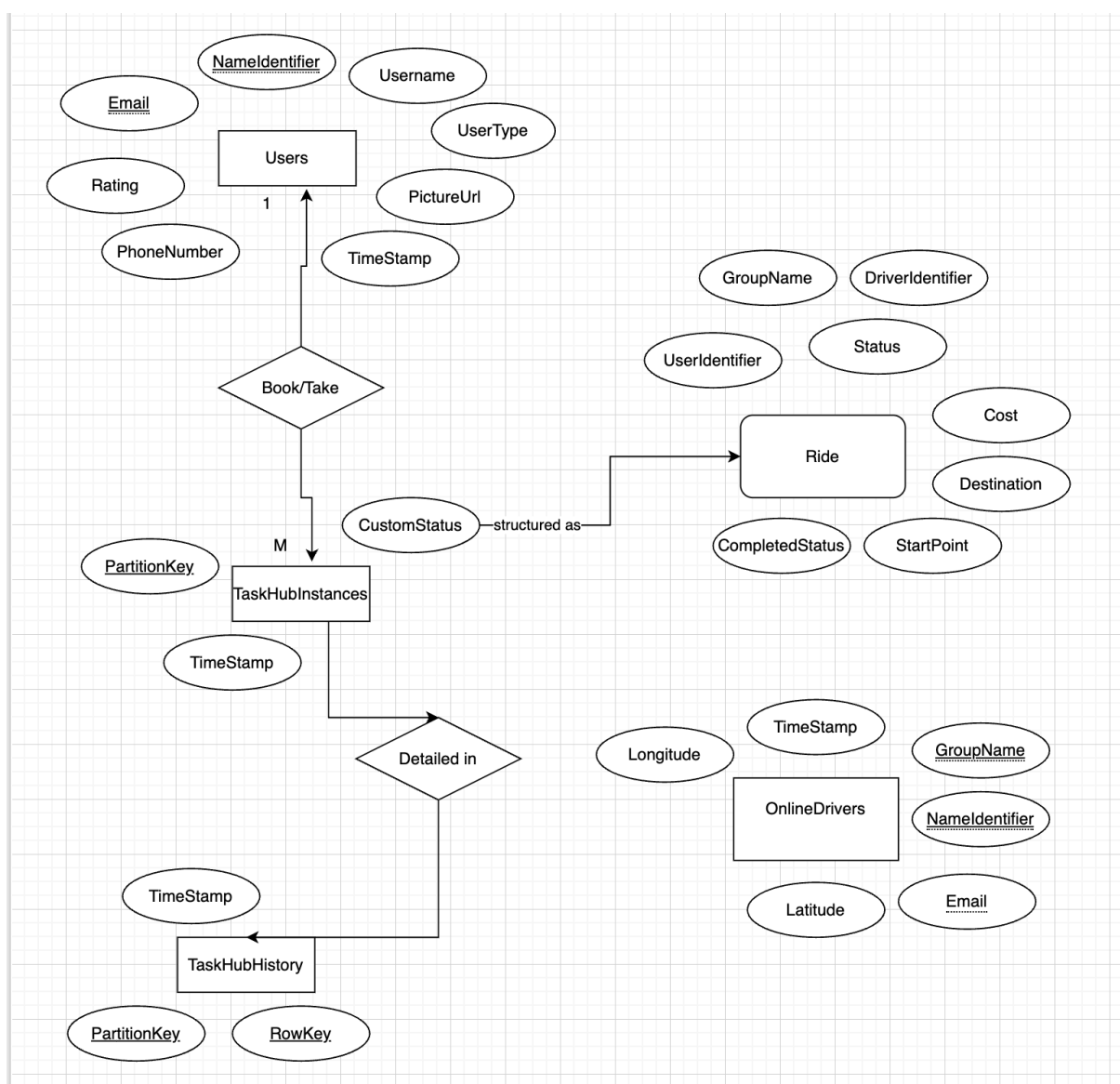


Fig. 5.9. Diagrama ER pentru baza de date.

Aplicația FastRide conține trei entități, fiecare reflectând o componentă critică a funcționalității

sistemului: utilizatorii, cursele și șoferii online.

Totul pornește de la entitatea *Users*, care stochează informațiile de bază ale fiecărui utilizator autentificat prin Google. Fiecare utilizator este identificat în mod unic prin *NameIdentifier* și *Email*, iar alături de acesta se păstrează date precum numele (*Username*), tipul de utilizator (*Driver*, *User* sau *Admin*), ratingul acumulat de-a lungul timpului, numărul de telefon și o eventuală poză de profil (*ImageUrl*).

În momentul în care un utilizator rezervă sau preia o cursă, se creează o relație între acesta și instanțele de orchestrare. Aici intervin tabelele *TaskHubInstances* și *TaskHubHistory*, care reflectă intern mecanismul de orchestrare al Azure Durable Functions. Fiecare instanță a unei funcții durabile este stocată în *TaskHubInstances*, iar starea sa (*CustomStatus*) este utilizată pentru a reflecta progresul unei curse (*Ride*). În paralel, *TaskHubHistory* păstrează detaliile cronologice ale execuției, istoricul exact al fiecărei acțiuni legate de instanța respectivă. Ambele entități sunt identificate prin *PartitionKey*, iar în cazul istoricului, și prin *RowKey*.

Starea unei curse este gestionată de modelul *Ride*, care este corelat cu *Users* prin câmpul *UserIdentifier*, indicând clientul care a inițiat cererea. Dacă un șofer acceptă cursa, acesta este reprezentat prin *DriverIdentifier*. Fiecare cursă este caracterizată de un status:

- *NewRideAvailable* - status-ul inițial al cursei;
- *GoingToUser* - șoferul acceptă cursa și navighează către client;
- *GoingToDestination* - șoferul merge spre destinație împreună cu pasagerul;
- *Finished* - cursa este finalizată cu succes;
- *Cancelled* - cursa a fost anulată;
- *None* - state-ul inițial.

Mereu după ce cursa se termină, aceasta se setează pe state-ul inițial datorită evitării de a complica infrastructura proiectului. Pentru a face vizibil în continuare statusul final al cursei, aceasta memorează și un *CompletedStatus*:

- *DriverNotFound* - nu s-a găsit niciun șofer pentru a finaliza cursa;
- *PaymentRefused* - clientul a refuzat plata (disponibilă doar cu cardul);
- *Cancelled* - cursa este anulată din orice alte motive;
- *Completed* - cursa a fost terminată cu succes.

În informațiile cursei se mai regăsesc și punctul de plecare (*StartPoint*), destinația (*Destination*) și costul asociat. *GroupName* este folosit pentru a lega această entitate de sesiunea real-time corespunzătoare, utilă pentru actualizări prin SignalR, astfel încât, doar șoferii din același grup cu clientul ce a inițiat cursa pot interacționa.

Pe lângă aceste entități persistente, aplicația păstrează în mod temporar o listă cu șoferii activi prin tabela *OnlineDrivers*. Fiecare șofer online este identificat prin *NameIdentifier* și asociat cu *GroupName*, astfel încât să poată fi notificat în timp real dacă există cereri în zona sa. Poziția sa curentă este păstrată prin coordonatele *Latitude* și *Longitude*, permițând localizarea sa pe hartă.

Azure Table Storage oferă performanță, scalabilitate și costuri reduse, iar modelul este suficient de flexibil pentru a susține dezvoltări ulterioare, cum ar fi introducerea plăților reale, recenziilor textuale sau programării curselor în avans.

5.3 Server-side

Toată logica aplicației este controlată de back-end. El este responsabil pentru procesarea curselor, și salvarea tuturor datelor în *Storage*.

Back-end-ul aplicației este construit folosind .NET și Azure Functions, oferind o arhitectură serverless, scalabilă și eficientă pentru gestionarea întregii logici din spatele aplicației.

Azure Durable Functions reprezintă o extensie a Azure Functions care permite definirea de fluxuri de lucru orchestrate - adică procese formate din mai mulți pași ce pot rula pe o perioadă mai lungă de timp, păstrând în același timp starea între execuții.

Spre deosebire de funcțiile serverless clasice (care sunt „stateless” – nu rețin nimic după ce se termină execuția), Durable Functions pot memora progresul, relua execuția exact de unde au rămas, și gestiona pași asincronici (de exemplu: așteptarea unui răspuns de la un șofer sau confirmarea unei curse).

Pentru a realiza asta, Azure păstrează în mod automat jurnalul execuției (event sourcing) într-un storage (de regulă Azure Storage), fără ca dezvoltatorul să se ocupe de salvarea sau reîncărcarea stării manual.

Prin această arhitectură, aplicația poate gestiona fluxuri complexe fără să piardă starea, fără să aibă probleme de sincronizare și fără să se bazeze pe baze de date tradiționale pentru urmărirea progresului. Acest lucru oferă fiabilitate, scalabilitate și claritate în logica aplicației.

Funcționarea aplicației se bazează pe modul în care partea de client, realizată în Blazor WebAssembly, comunică cu back-end-ul prin HTTP pentru cereri standard și prin SignalR pentru recepționarea în timp real a evenimentelor.

Structura proiectului Server, este cuprinsă din mai multe niveluri: function triggers, orchestrations, activities, services și repositories.

Function triggers sunt de 3 tipuri: *HttpTrigger*, *TimeTrigger* și *SignalRTrigger*.

HttpTriggers sunt folosite pentru comunicarea cu Client-side, și sunt requesturi de tipul *Get/Post* și *Put*.

```

31 [Authorize(UserRoles = [UserType.User, UserType.Driver, UserType.Admin])]
32 [Function(nameof(GetCurrentUserAsync))]
33 public async Task<IActionResult> GetCurrentUserAsync(
34     [HttpTrigger(AuthorizationLevel.Function, "get", Route = "user")] HttpRequest req)
35 {
36     _logger.LogInformation($"{nameof(GetUserAsync)} HTTP trigger function processed a request.");
37
38     var response = await _userService.GetUserAsync(new UserIdentifier()
39     {
40         NameIdentifier = req.HttpContext.User.Claims.Single(x => x.Type == "sub").Value,
41         Email = req.HttpContext.User.Claims.Single(x => x.Type == "email").Value
42     },
43     req.HttpContext.User.Claims.Single(x => x.Type == "name").Value); // Task<ServiceResponse<...>>
44
45     if (response.Success)
46     {
47         if (response.Response.PictureUrl != req.HttpContext.User.Claims.Single(x => x.Type == "picture").Value)
48         {
49             var update :ServiceResponse = await _userService.UpdateUserAsync(
50                 new UpdateUserPayload()
51                 {
52                     User = response.Response.Identifier,
53                     PhoneNumber = response.Response.PhoneNumber,
54                     PictureUrl = req.HttpContext.User.Claims.Single(x => x.Type == "picture").Value
55                 }); // Task<ServiceResponse>
56
57             if (!update.Success)
58             {
59                 return ApiServiceResponse.ApiServiceResult(update);
60             }
61         }
62     }
63 }

```

Fig. 5.10. Exemplu de metodă GET pentru HttpTrigger function.

Aceste request-uri sunt autorizate prin JWT token obținut de la Google. Astfel s-a creat un atribut custom, ce poate fi folosit doar pe metode, unde se pot defini o listă de *roles* ce pot accesa metoda respectivă. Ca și request, body-ul este încapsulat în modelul *HttpRequest* și poate fi deserializat astfel:

```

76 string requestBody;
77 using (var streamReader = new StreamReader(req.Body))
78 {
79     requestBody = await streamReader.ReadToEndAsync();
80 }
81
82 var request = JsonConvert.DeserializeObject<UserIdentifier>(requestBody);
83

```

Fig. 5.11. Deserializarea unui request într-un tip concret (ex: *UserIdentifier*).

Ca și răspuns, toate funcțiile HTTP întorc un *IActionResult* ce este mapat printr-o extensie a unui *ServiceResponse*. Acest model conține status-ul request-ului, mesajul de eroare (dacă există) și payload-ul response-ului. Toate service-urile returnează un *ServiceResponse* care, la fel, conține payload-ul, dacă este cu succes sau nu și mesajul de eroare împreună cu excepții. Dacă există mesaj de eroare atunci rezultatul este un *Bad Request (400)*, iar dacă există excepție atunci se traduce ca *Internal Server Error (500)*.

```
41 public static IActionResult ApiServiceResult<T>(ServiceResponse<T> serviceResponse)
42     where T : new()
43 {
44     if (serviceResponse.Success)
45     {
46         return new ObjectResult(serviceResponse.Response);
47     }
48
49     if (serviceResponse.Exception == null)
50     {
51         return new ContentResult
52         {
53             ContentType = MediaTypeNames.Text.Plain,
54             Content = serviceResponse.ErrorMessage,
55             StatusCode = (int)HttpStatusCode.BadRequest
56         };
57     }
58
59     return new ContentResult
60     {
61         ContentType = MediaTypeNames.Text.Plain,
62         Content = serviceResponse.ErrorMessage,
63         StatusCode = (int)HttpStatusCode.InternalServerError
64     };
65 }
```

Fig. 5.12. Maparea unui *ServiceResponse* la *IActionResult*.

Metodele HTTP disponibile pentru Client sunt:

- *GetCurrentUserAsync* pentru a lua informațiile despre user-ul de a inițiat requestul;
- *GetUserAsync* pentru a lua informațiile despre un user;
- *GetUsers*, disponibil doar pentru admini, pentru a vedea toți utilizatorii;
- *UpdateUserAsync* pentru a actualiza un user;
- *GetRidesByUserAsync* pentru a afișa cursele unui utilizator

Pentru a autoriza aceste endpoint-uri, în Azure Functions se pot înregistra *middleware* pentru a verifica token-ul. Astfel *AuthenticationMiddleware* și *AuthorizationMiddleware* se ocupă de autentificarea și autorizarea API-ului, luând JWT token-ul din *Authorization* header, și îl validează folosind API-ul de la Google.

```

42 private static async Task<bool> IsAccessTokenValidAsync(string accessToken)
43 {
44     using var client = new HttpClient();
45     var response =
46         await client.GetAsync(
47             requestUri: $"{{Environment.GetEnvironmentVariable("Google:Api")}}tokeninfo?access_token={{accessToken}}");
48
49     return response.IsSuccessStatusCode;
50 }

```

Fig. 5.13. Verificare *AccessToken* folosind Google API.

Pentru a face Google API conștient despre existența aplicației și token-ului pe care user-ul îl obține prin autentificarea pe Client, în Google Console, se creează un *Credential* pentru proiect. Redirect URIs și request browser sunt reprezentate de Client URL (de acolo se apelează autentificarea).

Client ID for Web application Delete

Name *
fast-ride
The name of your OAuth 2.0 client. This name is only used to identify the client in the console and will not be shown to end users.

Additional information

Client ID	380926423162-qve7d250c6r1ej1o1
Creation date	19 November 2024 a
Last used date	30 May 2025 (Note: 1

Authorised JavaScript origins ⓘ
For use with requests from a browser

URIs 1 *
https://localhost:7028
+ Add URI

Authorised redirect URIs ⓘ
For use with requests from a web server

URIs 1 *
https://localhost:7028/authentication/login-callback
+ Add URI

Client secrets
If you are in the process of changing client secrets, you can manually

Client secret	GOCSPX-Nm8likVZhDzVn4t-ifpr
Creation date	19 November 2024 at 10:59:15
Status	Enabled

+ Add secret

Fig. 5.14. Configurația Google Credentials.

TimeTrigger functions reprezintă funcțiile ce se auto-aplează la o perioadă de timp configurată. FastRide folosește un acest tip de funcție pentru a interoga din 5 în 5 secunde tabelul generat de Durable Orchestration în scopul de a găsi curse ce se află în progres și de a le notifica Client-ului prin SignalR.

```

59 [Function(nameof(NotifyUserStateAsync))]
60 [SignalROutput(HubName = SignalRConstants.HubName)]
61 3 usages Madalin Popescu
62 public async Task<List<SignalRMessageAction>> NotifyUserStateAsync(
63     [TimerTrigger(schedule: "*/5 * * * * * (Every 5 seconds)")] TimerInfo timerInfo,
64     [DurableClient] DurableTaskClient client)
65 {
66     _logger.LogInformation($"{nameof(NotifyUserStateAsync)} timer trigger function started.");
67
68     var instances :AsyncPageable<OrchestrationMetadata> = client.GetAllInstancesAsync(new OrchestrationQuery()
69     {
70         Statuses =
71         [
72             OrchestrationRuntimeStatus.Pending,
73             OrchestrationRuntimeStatus.Running,
74             OrchestrationRuntimeStatus.Suspended
75         ]
76     });
77     var rides :List<Ride> = await MapRidesAsync(instances);
78     return rides.Select(ride => new SignalRMessageAction( target: SignalRConstants.ServerNotifyState)
79         { Arguments = [ride], GroupName = ride.Id, }).ToList(); // List<SignalRMessageAction>
80 }

```

Fig. 5.15. TimeTrigger function ce interoghează Durable tables și notifică utilizatorii despre curse lor ce sunt în desfășurare.

Această funcție caută orchestrările ce se află în *Pending*, *Running* sau *Suspended*, citește proprietatea *CustomStatus* unde se regăsesc informațiile complete ale cursei și le returnează pentru fiecare grup de utilizatori ce se află în parcursul cursei respective (în acest caz, *GroupName* devine *ride.Id* - ID-ul cursei).

SignalRTrigger sunt function-urile ce ascultă la un canal WebSocket pentru SignalR, dar sunt și capabile să returneze un mesaj către acest canal. Acestea sunt folosite pentru comunicarea în timp real cu Client-ul și au rolul de a notifica utilizatorii despre diferite acțiuni ce trebuie să le facă (ex: să plătească, să accepte o cursă, etc). Acest tip de funcții folosesc un *eventType* ca și identificator pentru a notifica utilizatorii ce sunt conectați la canal. Utilizatorii pot fi grupați în grupuri pentru a trimite mesaje doar către un anumit grup, însă se pot trimite mesaje și direct către un utilizator specific.

FastRide folosește grupurile pentru a grupa utilizatorii per orașe, astfel cei din Craiova nu pot vedea și nu pot primi sau trimite mesaje de la sau către cei din București, spre exemplu. Grupurile se mai folosesc și pentru a identifica doi utilizatori ce se află într-o cursă, astfel în momentul în care o cursă se inițiază, atât șoferul cât și clientul, părăsesc automat grupul din care fac parte (o combinație dintre țară, județ și oraș) și se înscriu la grupul ce se formează din ID-ul cursei generat la creare.

Pentru a se conecta la canalul de SignalR și pentru a se înscrie într-un grup sau pentru a părăsi un grup, Client-ul trebuie să apeleze câteva metode care indică aceste *actions*.

```

21     [Function(nameof(JoinUserToGroup))]
22     [SignalROutput(HubName = SignalRConstants.HubName)]
23     2 usages Madalin Popescu +1
24     public async Task<SignalRGroupAction> JoinUserToGroup(
25         [SignalRTrigger(SignalRConstants.HubName, category: "messages", event: SignalRConstants.ClientJoinUserToGroup, "userId",
26             "groupName")]
27         SignalRInvocationContext invocationContext,
28         [DurableClient] DurableTaskClient client, string userId, string groupName)
29     {
30         var instances :AsyncPageable<OrchestrationMetadata> = client.GetAllInstancesAsync(new OrchestrationQuery()
31         {
32             Statuses =
33             [
34                 OrchestrationRuntimeStatus.Pending,
35                 OrchestrationRuntimeStatus.Running,
36                 OrchestrationRuntimeStatus.Suspended
37             ]
38         });
39         await foreach (var instance :OrchestrationMetadata in instances)
40         {
41             // ...
42         }
43     }
44 }
45
46 [Function(nameof(LeaveUserFromGroup))]
47 [SignalROutput(HubName = SignalRConstants.HubName)]
48 2 usages Madalin Popescu +1
49 public SignalRGroupAction LeaveUserFromGroup(
50     [SignalRTrigger(SignalRConstants.HubName, category: "messages", event: SignalRConstants.ClientLeaveUserFromGroup, "userId",
51         "groupName")]
52     SignalRInvocationContext invocationContext, string userId, string groupName)
53 {
54     return new SignalRGroupAction(SignalRGroupActionType.Remove)
55     {
56         GroupName = groupName,
57         UserId = userId
58     };
59 }
60 }

```

Fig. 5.16. Înscrierea și ștergerea unui utilizator din grup.

Pentru ca server-ul SignalR să înregistreze acțiunile aferente, Server-ul trebuie să returneze un *SignalRGroupAction*. Pentru restul notificărilor se folosește *SignalRMessageAction*.

Conectarea la SignalR de către utilizator și notificarea că a fost conectat sau deconectat cu succes se face prin metodele din imaginile din fig. 5.17. Astfel, în momentul în care un șofer este conectat sau deconectat, se poate actualiza tabela *OnlineDrivers*.

Se observă faptul că este nevoie de un identificator al utilizatorului, acesta fiind *NameIdentifier*-ul din storage, care este totodată și claim-ul *sub* din JWT token-ul de la Google, doar că hash-uit (SHA256). Hash-ul se aplică pentru a obține o structură de *GUID*.


```

27 [Function(name: "negotiate")]
    1 usage 2 DizzYForester +1
28 public IActionResult Negotiate(
29     [HttpTrigger(AuthorizationLevel.Function, params methods: "post", Route = "negotiate")]
30     HttpRequest req,
31     [SignalRConnectionInfoInput(HubName = SignalRConstants.HubName, UserId = "{query.userId}")]
32     SignalRConnectionInfo signalRConnectionInfo)
33 {
34     _logger.LogInformation("Negotiate request received");
35     madalin-popescu, 04.03.2025, 17:08 • fix user ID
36     return new ObjectResult(signalRConnectionInfo);
37 }

39 [Function(name: "onconnected")]
40 [SignalROutput(HubName = SignalRConstants.HubName)]
    1 usage 2 DizzYForester +1
41 public async Task<SignalRMessage> OnConnected(
42     [SignalRTrigger(hubName: SignalRConstants.HubName, category: "connections", @event: "connected")]
43     SignalRInvocationContext context)
44 {
45     return new SignalRMessage
46     {
47         Target = "Connected"
48     };
49 }

51 [Function(name: "ondisconnected")]
52 [SignalROutput(HubName = SignalRConstants.HubName)]
    1 usage 2 DizzYForester +1
53 public async Task<SignalRMessage> OnDisconnected(
54     [SignalRTrigger(hubName: SignalRConstants.HubName, category: "connections", @event: "disconnected")]
55     SignalRInvocationContext context)
56 {
57     await _driversService.DeleteOnlineDriverAsync(context.UserId);
58
59     return new SignalRMessage
60     {
61         Target = "Disconnected"
62     };
63 }
64 }

```

Fig. 5.17. *Negotiate*, conectarea și deconectarea unui utilizator de la conexiune SignalR.

Orchestrations sunt Azure Functions ce pot rula mai mult de 5 minute, ce își păstrează state-ul în tabele, astfel încât indiferent de ce se întâmplă cu runtime-ul aplicației, acestea își continuă execuția fără a relua tot procesul.

În proiect, există o singură funcție de acest gen, ce se ocupă cu flow management-ul unei curse. Astfel, ea primește ca și input, un request ce conține un *User* (clientul), locul de unde se inițiază cursa și destinația dorită. La fiecare pas, aceste informații se actualizează și se salvează în coloana specială a Durable Orchestrations, *CustomStatus*, totodată, modificând și status-ul cursei.

```

36     var input = context.GetInput<NewRideInput>();
37     context.SetCustomStatus(input);

```

Fig. 5.18. Salvarea în *CustomStatus* a request-ului inițial.

Mai departe, aplicația calculează prețul cursei și trimite către utilizator un request de confirmare a cursei. Pentru a aștepta răspuns, Durable Orchestrations folosesc evenimente pe care le așteaptă:

```

113     await context.CallActivityAsync(
114         nameof(SendPriceCalculationActivity),
115         new SendPriceCalculationActivityInput
116         {
117             InstanceId = context.InstanceId,
118             UserId = input.User.NameIdentifier,
119             Destination = input.Destination,
120             StartPoint = input.StartPoint,
121         }); // Task
122
123     var priceAccepted =
124         await context.WaitForExternalEvent<string>(SignalRConstants.ClientSendPriceCalculation);
125     var accepted:double = double.Parse(priceAccepted);
126     return accepted;

```

Fig. 5.19. Trimiterea confirmării către utilizator și așteptarea răspunsului.

Pentru a calcula prețul se folosește următoarea formulă (distanța dintre 2 puncte în spațiu, estimarea timpului de parcurgere cu un timp comun și prețul configurat pentru fiecare kilometru și minut, totul adunat cu o sumă inițială):

```

public double CalculatePricePerDistance(double distanceInKm, double durationInMinutes)
{
    logger.LogInformation("Calculate price for current distance!");
    return _basePrice + _pricePerKm * distanceInKm + _pricePerMinute * durationInMinutes;
}

```

Fig. 5.20. Calcularea prețului cursei.

Dacă s-a acceptat, urmează să se inițieze plata, astfel, Server-ul se ocupă cu crearea unui *PaymentIntent* ce se poate folosi pentru a se realiza plata cu succes. Fără această intenționare, Stripe nu știe să autentifice request-ul de plată.

```

27     var stripe = new StripeClient(apiKey: Environment.GetEnvironmentVariable("Stripe:SecretKey"));
28     var paymentIntentService = new PaymentIntentService(stripe);
29     var paymentIntent = await paymentIntentService.CreateAsync(new PaymentIntentCreateOptions
30     {
31         Amount = (long)(input.Price * 100), // Amount in cents
32         Currency = "ron",
33         PaymentMethodTypes = ["card"],
34     }); // Task<PaymentIntent>

```

Fig. 5.21. Crearea *PaymentIntent*-ului prin Stripe.

Mai departe, după ce plata s-a procesat cu success, începe să se caute un șofer pentru cursă. Algoritmul de căutare se folosește de tabelul *OnlineDrivers* pentru a identifica poziția fiecărui șofer și a îi calcula distanța de la client la acesta. Șoferii se iau în ordine până se găsește unul sau niciunul. Dacă șoferul nu acceptă cursa în timp de 35 secunde, se consideră ca și respinsă. Pentru a face acest lucru posibil, se creează o listă de *Task*-uri, returnată de *WaitForExternalEvent*-urile, și așteaptă ca cel puțin unul dintre task-uri să se completeze. În funcție de task-ul completat, se alege acțiunea de a merge mai departe sau de a căuta alt șofer.

```

192     var clientResponseTask =
193         context.WaitForExternalEvent<DriverAcceptResponse>(SignalRConstants.ClientDriverAcceptRide);
194
195     var timeoutTask = context.CallActivityAsync<DriverAcceptResponse>(nameof(DelayActivity),
196         new DelayActivityInput()
197         {
198             Seconds = 35,
199             DriverIdentifier = driver.Identifier.NameIdentifier,
200         });
201
202     var completedTask = await Task.WhenAny(clientResponseTask, timeoutTask);
203
204     if (completedTask == clientResponseTask)
205     {
206         if ((await completedTask).Accepted)
207         {
208             return (await completedTask).UserId;
209         }
210     }

```

Fig. 5.22. Așteptarea răspunsului de la șofer și cronometrarea timpului de acceptare.

La orice pas, dacă utilizatorul anulează fie confirmarea, fie plata, fie nu se găsește un șofer, cursa intră într-un status terminal, setând proprietatea *CompletedStatus* cu un status aferent acțiunii.

```

254     private static async Task CancelWorkflow(TaskOrchestrationContext context, NewRideInput input,
255         CompleteStatus completeStatus)
256     {
257         input.Status = InternRideStatus.Cancelled;
258         input.CompleteStatus = completeStatus;
259         context.SetCustomStatus(input);
260         await context.CallActivityAsync(⚡ nameof(DelayActivity),
261             new DelayActivityInput()
262             {
263                 Seconds = 25
264             }); // Task
265         input.Status = InternRideStatus.None;
266         context.SetCustomStatus(input);
267         await context.CallActivityAsync(⚡ nameof(DelayActivity),
268             new DelayActivityInput()
269             {
270                 Seconds = 35
271             }); // Task
272     }

```

Fig. 5.23. Anularea cursei dacă plata nu s-a procesat, a fost intenționat anulată sau nu s-a găsit un șofer.

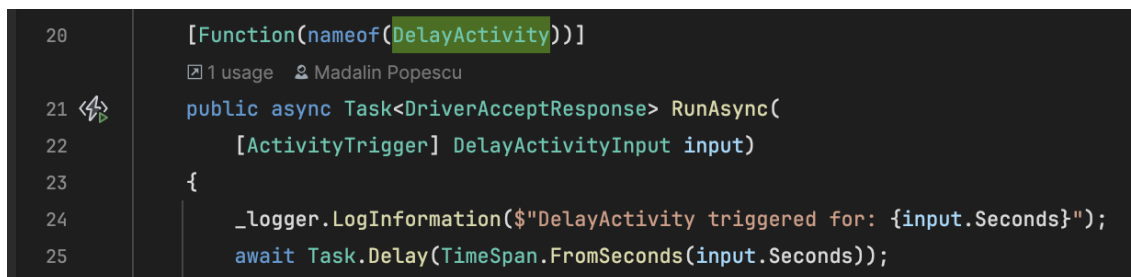
Delay-urile au rolul de a face posibilă așteptarea ca și utilizatorul să primească aceste informații, altfel totul ar fi instant, iar utilizatorul nu ar vedea niciun feedback.

Funcțiile durabile de orchestrare trebuie să fie deterministice, astfel încât dacă aplicația repornește sau un proces de resetare intervine, codul să rezulte același rezultat. Pentru asta, trebuie evitate în contextul orchestration-ului să se genereze valori random sau să se apeleze service-uri. Totuși, este nevoie să se apeleze service-uri precum calcularea pretului cursei sau căutarea unui șofer. Pentru a face posibil acest lucru, intervin activitățile.

Activity functions sunt funcții ce au rol de a menține pasul la care orchestration-ul rulează. Pentru fiecare acțiune necesară în orchestration, există o activitate, astfel:

- *CancelRideActivity* - trimite cererea de anulare a cursei către toți participanții;
- *DelayActivity* - suspendă orchestration-ul pentru n secunde;
- *FindDriverActivity* - caută un șofer pentru cursa trimisă. Totodată, se așteaptă să primească și lista de șoferi ce au fost excluși, datorită faptului că au respins deja cererea;
- *NotifyDriverTimeoutActivity* - notifică șoferul că perioada de acceptare a cursei a expirat;
- *SendPaymentIntentActivity* - creează *PaymentIntent*-ul și îl trimite către client;
- *SendPriceCalculationActivity* - calculează prețul cursei și îl notifică pe client despre acesta;
- *SendRatingRequestActivity* - la finalul cursei, clienții trebuie să ofere un notă șoferului, această activitate ocupându-se de această sarcină;

- *SendRideToDriverActivity* - notifică șoferul despre cursă, acesta fiind nevoit să decidă dacă acceptă cursa sau nu.



```
20 [Function(nameof(DelayActivity))]  
    1 usage 2 Madalin Popescu  
21 public async Task<DriverAcceptResponse> RunAsync(  
22     [ActivityTrigger] DelayActivityInput input)  
23 {  
24     _logger.LogInformation($"DelayActivity triggered for: {input.Seconds}");  
25     await Task.Delay(TimeSpan.FromSeconds(input.Seconds));
```

Fig. 5.24. Declararea activității ce pune orchestration-ul să aștepte n secunde.

Pentru a structura logica, separat de activități, se folosesc servicii. *Services* sunt instanțe de clase înregistrate în contextul aplicației pentru a putea fi injectate mai târziu, și folosite să execute diferită logică. Spre exemplu, calcularea distanței dintre două puncte este o metodă dintr-un service înregistrat *Scoped* (se generează o instanță nouă la fiecare request inițiat). Alte servicii ce se regăsesc în proiect sunt pentru calcularea prețului, pentru a găsi șoferi în zonă apropiată de un client și de a intermedia operațiile *CRUD* (Create Read Update Delete) pentru utilizatori și curse. Toate serviciile folosesc o structură comună de returnare a datelor, răspunsul fiind încapsulat într-un *ServiceResponse* ce conține payload-ul, un mesaj de eroare sau o excepție, dacă există și dacă totul s-a executat cu succes sau nu. Acest tip de răspuns ajută la handle-uirea răspunsului și încapsularea lui într-un *ApiResponse*.

Configurarea proiectului se face prin *Environment Variables*, local, definindu-se prin *local.settings.json*. Această configurație este alcătuită din:

- Google Credentials - pentru a reuși validarea token-ului de acces ce vine pe request;
- Distance Configuration - ce conțin prețurile pentru un kilometru, un minut, prețul inițial și viteza medie de calcul pentru distanță;
- Stripe Secret Key - folosit pentru generarea intenționării de plată;
- Storage Connection String - conexiunea la Azure Storage.
- SignalR Connection String - conexiunea la server-ul SignalR

Microsoft, loghează fiecare activitate ce se execută, iar în caz de deployment pe o infrastructură ce permite logging (spre exemplu Azure), este necesar să se specifice ce anume să se logheze pentru a nu se ajunge la costuri mari

```
28 .ConfigureLogging((context, b) =>
29 {
30     //https://github.com/Azure/azure-functions-host/issues/8973#issuecomment-1890784686
31     b.AddConsole();
32     b.AddFilter(category: "", LogLevel.Information);
33     b.AddFilter(category: "Azure.Core", LogLevel.Warning);
34 }) // IHostBuilder
```

Fig. 5.25. Filtrarea log-urilor din Azure.

5.4 Client-side

Pentru a putea afișa toate aceste informații și a notifica utilizatorul despre toate procesele de se întâmplă în spate, front-end-ul stă la dispoziția utilizatorului și este singura componentă ce interacționează activ cu acesta.

Client-side-ul se ocupă cu afișarea informațiilor procesate pe Server într-un mod cât mai prietenos și ușor de înțeles.

La bază acestuia, stă Blazor WASM, ce funcționează ca o aplicație instalată, fără a avea nevoie de o componentă de server sau să instanțieze două componente de comunicare. Toate asset-urile se descarcă în browserul utilizatorului și poate funcționa fără conexiune la internet.

Folderul *wwwroot* este rădăcina conținutului static al aplicației Blazor WebAssembly. Tot ce se află în acest folder (sau în subfolderele lui) este expus publicului și servit direct browserului, fără procesare pe server. Aici se regăsește configurația proiectului pentru debugging, stilurile CSS, logica JavaScript, dar și datele mock pentru debugging.

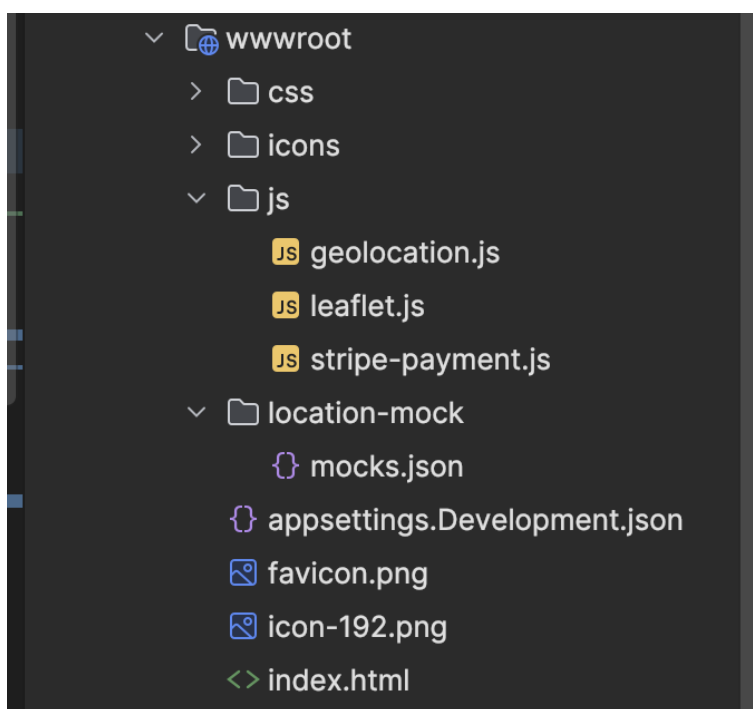


Fig. 5.26. Conținutul folderului *wwwroot*.

Configurația proiectului conține informațiile de autentificare pentru Google (*ClientId*, *Authority*, *Scopes*, etc), setări predefinite pentru hartă, conexiunea cu server-ul și *PublishKey*-ul pentru Stripe.

Sistemul de hărți este susținut de Leaflet, combinat cu modulul Routing Machine pentru calcularea traseului. Există un Nuget Package pentru Blazor însă suportul pentru Routing nu vine inclus. De aceea se folosește librăria așa cum vine ea, fără alte pachete instalate, doar prin JavaScript.

Pentru a face legătura dintre Blazor (C#) și JavaScript, există JSInterop. *JSInterop* (JavaScript Interop) este mecanismul prin care aplicațiile Blazor pot comunica cu JavaScript. Prin JSInterop, codul scris în C# poate apela funcții JavaScript, iar JavaScript poate la rândul său să invoce metode din C#. Acest mecanism este esențial atunci când Blazor nu oferă suport direct pentru anumite funcționalități disponibile doar prin API-urile browserului sau prin biblioteci JavaScript externe. Interacțiunea este asincronă și se realizează prin intermediul interfeței *IJSRuntime*, fiind disponibilă atât în Blazor WebAssembly, cât și în Blazor Server. [6]

Urmând pașii de folosire a hărții Leaflet, s-a creat un script în JS, pentru a putea fi apelat din Blazor și a inițializa harta. Totul s-a încapsulat într-o componentă *Map.razor*. După ce componenta se randează, se construiește harta, care suprascrie un *div* tag cu ID-ul *map*, instanțiază un *.NET Object Reference* (pentru a putea fi apelat codul .NET din JS) și se folosește de informațiile despre locația curentă, zoom-ul pe hartă și alte setări pentru configurare.

```
37 map = L.map('map').setView([startLat, startLng], zoom);
38
39 L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
40     maxZoom: 19,
41     attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
42 }).addTo(map);
43
44 leafletAddUser(userId, startLat, startLng, img);
45
46 map.on('click', onMapClick);
```

Fig. 5.27. Inițializarea hărții după ce componenta s-a randat.

Orice iconiță adăugată pe hartă reprezintă un *marker* cu o imagine ca și înfățișare. Pentru a pune un *pin*, spre exemplu, există o metodă numită *SetPinLocationAsync* ce adaugă un *marker* pe hartă cu *asset-ul de pin*. Toate *asset-urile* se află în *wwwroot/icons*: *pin*, *human*, *driver* și *currentCar* (ce indică că atât clientul cât și șoferul se află în aceeași mașină). Metodele *DrawRoute* și *AddUser/Driver* ajută la construirea traseului și adăugarea unui utilizator pe hartă.

```
31     private Dictionary<string, string> _assets = new()  
32     {  
33         { "driver", "icons/driver.png" },  
34         { "human", "icons/human.png" },  
35         { "currentCar", "icons/currentCar.png" },  
36         { "pin", "icons/pin.png" }  
37     };
```

Fig. 5.28. Dicționarul ce face legătura dintre tipul de marker și imaginea de îl reprezintă.

Același mecanism de afișare îl folosește și dialog-ul pentru a introduce cardul bancar. Se atașează input-urile de elementul *div* cu ID-ul *payment-element*. Acesta validează un card bancar folosind API-ul Stripe, iar pentru a putea autoriza acest request, se folosește de *PublishKey* din dashboard-ul Stripe.

```
13     const paymentElement : payment = elements.create("payment", paymentElementOptions);  
14     paymentElement.mount("#payment-element");
```

Fig. 5.29. Atașarea input-ului pentru card bancar de elementul cu ID-ul *payment-element*.

Toate tranzacțiile sunt se pot vedea în platfomă, dacă au fost cu succes, ce card a fost folosit sau de către ce customer a fost plătit. Fiind în modul de testare, se poate folosi doar cardul cu numărul: 4242 4242 4242 4242, restul informațiilor trebuind să fie doar corecte, nu și existente.

The screenshot displays the Stripe dashboard interface. At the top, there's a 'Test mode' header with a note 'You're using test data'. Below this is a navigation bar with tabs: Overview, Webhooks, Events, Logs, Errors, Inspector, Blueprints, and Shell. The main content area is divided into several sections:

- API keys:** Shows 'Standard keys' with options for 'Publishable key' and 'Secret key'. It displays two test keys: 'pk_test_...ZCEV' and 'sk_test_...qzHm'. A 'Manage API keys' link is present.
- API requests:** Shows '0 total' and '0 failed' requests. A 'View all requests' link is available.
- API versions:** Shows two versions: '2025-05-28.basil' (marked as 'Latest') and '2024-09-30.acacia' (marked as 'Default'). An 'Upgrade available' badge and an 'Upgrade' link with an arrow are also present.
- Transactions:** A section with a recommendation to use AI for writing Stripe code. It includes filters for 'All' (139), 'Succeeded' (108), 'Refunded' (0), and 'Disputed' (0). Below the filters are tabs for 'Date and time', 'Amount', 'Currency', 'Status', 'Payment method', and 'More filters'. A table lists transactions with columns for Amount, Payment method, Description, Customer, and Date.

Amount	Payment method	Description	Customer	Date
RON 8.94	Visa	pi_3RTzdSCpLPiMeksm0LGtRk03	dizzyforester@gmail.com	29 May,
RON 8.07	Visa	pi_3RTzbYCpLPiMeksm0v3aU0Hw	dizzyforester@gmail.com	29 May,
RON 4.50	Visa	pi_3RTzVFCpLPiMeksm1CRqw2Ld	dizzyforester@gmail.com	29 May,

Fig. 5.30. Configurarea Stripe în dashboard-ul platformei și istoricul tranzacțiilor.

Pentru transmiterea live a locației curente de către utilizatori și de a reuși să se expună pe hartă locul exact unde se află utilizatorul, se folosește un *Background Service* ce rulează în constant și face request de locație către JavaScript. Background Service-ul face acest tip de request din trei în trei secunde, folosind un *PeriodicTimer*.

```

51     public async Task StartExecutingAsync()
52     {
53         if (_running) return;
54
55         await SaveCurrentGeolocationAsync();
56
57         _running = true;
58         _cts = new CancellationTokenSource();
59         _timer = new PeriodicTimer(period: TimeSpan.FromSeconds(3));
60         while (await _timer.WaitForNextTickAsync(_cts.Token))
61         {
62             await HandleTimerAsync();
63         }
64     }

```

Fig. 5.31. Execuția din 3 în 3 secunde a Background Service-ului.

Pentru a reuși să se obțină locația curentă, codul din JavaScript nu returnează instant rezultatul, ci apelează un *callback method*. Astfel, pentru a reuși să se aștepte acest răspuns în .NET, se folosește *TaskCompletionSource* ce așteaptă pentru un *event*, mai exact ca acesta să își schimbe state-ul și să i se seteze un rezultat.

```

48     var tcs = new TaskCompletionSource<Geolocation>();
49
50     Func<Geolocation, ValueTask> coordinatesChangedHandler = null;
51     coordinatesChangedHandler = (geolocation) =>
52     {
53         tcs.SetResult(geolocation);
54
55         CoordinatesChanged -= coordinatesChangedHandler;
56
57         return ValueTask.CompletedTask;
58     };
59
60     CoordinatesChanged += coordinatesChangedHandler;
61
62     await RequestGeoLocationAsync();
63
64     var geolocation = await tcs.Task;

```

Fig. 5.32. *TaskCompletionSource* așteptând ca scriptul de JavaScript să trigger-uiască event-ul *CoordinatesChanged*

Pentru reuși să se păstreze acest tip de informații (locația curentă, status-ul curent al cursei, grupul din care face parte utilizatorul) se folosesc *State*-urile. Un *State* este un service

care nu necesită o interfață deoarece nu este făcut să handle-uiască logică, ci doar să păstreze valoarea unor proprietăți pe parcursul execuției. Ele se înregistrează *Scoped* pentru a se reseta instanța în momentul unui request nou (adică dacă alt utilizator deschide site-ul sau dă refresh), astfel valorile unui utilizator nu o să interfereze cu ale celorlalți.

Grupul din care face parte utilizatorul, despre care s-a menționat în secțiune de Server, se calculează aici deoarece Client-ul este singurul loc unde se poate detecta locația utilizatorului. Grupul este format din oraș, județ și țară și se poate afla folosind service-ul *IUserService*. Dacă utilizatorul se află într-o cursă, atunci service-ul returnează ID-ul cursei ca și grup pentru a reuși să se facă o conexiune doar între client și șofer.

Pentru informații precum: detectarea adresei în text, folosind geolocația, se folosește service-ul *ILocationService* ce folosește API-ul de la *OpenStreetMap*.

Pe lângă componenta hărții, mai există componentele ce implementează dialog-urile pentru rating, acceptarea cursei, plățirea cursei și componente precum *loading screen*-ul, afișarea status-ului cursei (dacă există una în progres) și butoane (pentru a porni o cursă sau a deschide dialog-ul de permite acceptarea uneia).

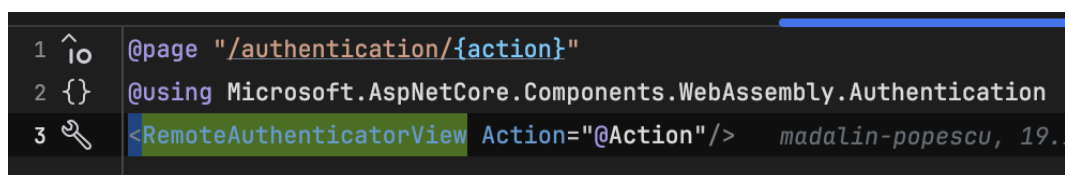
Pagina principală conține harta, un search pentru adrese, ce interoghează API-ul de la *OpenStreetMap* cu query-ul introdus de utilizator și returnează o listă de sugestii.

Pagina *History* conține istoricul curselor cu informații despre cursă precum prețul, data, dar și locația.

Adminii beneficiază de o pagină în plus unde pot vedea toți utilizatorii și le pot modifica informații precum numărul de telefon sau rolul.

Informațiile precum, numele sau imaginea sunt informații ce vin de pe contul Google al utilizatorului și acesta este redirecționat către pagina Google Accounts pentru a putea actualiza ce dorește. Toate acestea reprezentând și butoanele *nav-menu*-ului.

Autentificarea se face prin Google, astfel utilizatorul, dacă nu este autentificat, este redirecționat către pagina de autentificare ce face un *remote authentication*.



```
1 @page "/authentication/{action}"
2 @using Microsoft.AspNetCore.Components.WebAssembly.Authentication
3 <RemoteAuthenticatorView Action="@Action" /> madalin-popescu, 19.1
```

Fig. 5.33. Pagina de autentificare ce redirecționează utilizatorul către Google Login.

Pentru a putea autoriza butoane sau pagini, se folosește atributul *Authorize*. Acesta se folosește de un *AccountClaimsPrincipalFactory* și un *RemoteUserAccount* pentru a ști ce roluri se află pe token-ul utilizatorului.

```

40 builder.Services.AddOidcAuthentication<RemoteAuthenticationState,
41     CustomUserAccount>(configure: options => { builder.Configuration.Bind(key: "Google", options.ProviderOptions); })
42     .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState, CustomUserAccount, CustomUserFactory>();
43
1  ^o @attribute [Authorize(Roles = "Admin")]
2    @page "/admin"

```

Fig. 5.34. Înregistrarea autentificării și folosirea autorizării în aplicație.

Layout-ul aplicației este cuprins dintr-un *navigation bar*, dialog-ul ce afișează status-ul cursei curente, loading screen-ul ce ascultă la State-ul *OverlayState*, dar și content-ul propriu zis al paginii curente.

Testarea unui asemenea proiect este dificil deoarece nimeni nu își dorește să se plimbe prin oraș pentru a îndeplini *test-case*-urile. Schimbarea locației unui dispozitiv fără un VPN este dificilă deoarece trebuie "păcălit" mai întâi browser-ul și apoi codul. De aceea, se folosesc date de testare, mock-uite, astfel utilizatorii cu un anumit identificator pot fi localizați în locuri diferite datorită fișierului *mocks.json*.

```

1  {
2    "d6088988-fe33-fd31-87b8-d42198b8d245": [
3      {
4        "Latitude": 44.30686316506801,
5        "Longitude": 23.81029470751658
6      },
7      {
8        "Latitude": 44.307325595681725,
9        "Longitude": 23.810131025094883
10     },

```

Fig. 5.35. Fișierul JSON ce conține coordonatele mock-uite pentru utilizatori.

Aceste date sunt preluate de către Background Service la fiecare trei secunde și sunt parcurse una câte una. În plus, metoda *HandleGeolocationMockAsync* interacționează cu datele acestui fișier pentru a putea face posibilă staționarea utilizatorilor sau resetarea index-ului de iterație prin fișier.

Pentru lucrarea curentă, s-a pregătit un singur set de date, astfel punctul de plecare se află pe Strada Anului 1848, Craiova, iar punctul de sosire este pe Strada Târgului, Craiova.

5.5 Instalarea aplicației

Pentru a putea folosi aplicația trebuie instalate mai multe componente:

- *Docker* - pentru storage
- *dotnet tool* - pentru comenzi în terminal și compilarea .NET;
- *Azurite* - emulatorul pentru Azure Storage;
- *SignalR Emulator* - emulatorul pentru server-ul SignalR;
- *Azure Functions Tools* - pentru pornirea proiectului Azure;
- *ngrok/Visual Studio sau Rider* - în caz de deployment sau doar un IDE pentru rularea codului.

Toate acestea susțin fiecare componentă din proiect.

Primul pas este să se obțină codul, fie din surse, fie de pe GitHub. Mai apoi se pot instala componentele.

Trebuie instalat *Docker* deoarece *Azurite* vine cu foarte multe fișiere temporale și există riscul ca mașina să rămână fără memorie.

Următorul pas reprezintă deschiderea unui *Terminal*, și să se navigheze către folderul *Docker* din proiect, folosind comanda:

```
cd C:\Work\FastRide\fast-ride\Run\Docker
```

Apoi trebuie rulată următoarea comandă pentru a instala *Azurite*:

```
docker-compose up -d
```

Dacă există eroare privind comanda *docker-compose*, atunci acesta trebuie instalat. Urmează să se instaleze *dotnet tool* și să se instaleze *SignalR Emulator*-ul prin comanda:

```
dotnet tool install -g Microsoft.Azure.SignalR.Emulator
```

Mai departe trebuie deschis *Docker*-ul și pornit *Azurite*, apoi rularea comenzii:

```
cd C:\Work\FastRide\fast-ride\Run\Docker  
asrs-emulator start
```

Pentru a rula codul, fie se deschide un IDE cu ambele soluții și se rulează, fie se rulează din terminal cu beneficiul de a publica în rețea aplicația. Pentru terminal trebuie rulate următoarele comenzi:

```
cd C:\Work\FastRide\fast-ride\FastRide.Server  
func start
```

```
ngrok http http://localhost:7102 # pentru ngrok (din ngrok.exe)
```

Dacă se folosește *ngrok* atunci trebuie actualizată configurația din Client pentru *SDK*-ul server-ului.

```
cd C:\Work\FastRide\fast-ride\FastRide.Client  
dotnet run --urls "http://localhost:7028"  
ngrok http https://localhost:7028 # pentru ngrok (din ngrok.exe)
```

6. CONCLUZII

Lucrarea a avut ca obiectiv dezvoltarea unei aplicații de tip ride-sharing, FastRide, care gestionează într-un mod eficient relația dintre clienți și șoferi. Aplicația pune accent pe simplitate, stabilitate și integrare între mai multe tehnologii actuale, reușind să acopere un flux funcțional complet: de la autentificarea utilizatorului până la vizualizarea unei curse finalizate.

Pe parcursul realizării proiectului, s-a urmărit nu doar implementarea funcționalităților de bază, ci și înțelegerea a modului în care diferite componente software pot comunica între ele. Utilizarea Blazor WebAssembly a oferit o experiență fluidă în browser, în timp ce Azure Durable Functions a permis gestionarea logicii din spatele aplicației într-un mod scalabil și bine structurat. Componentele precum SignalR, Leaflet, Stripe și integrarea cu contul Google au completat la integritatea aplicației, transformând-o într-un produs și serviciu pregătit pentru utilizarea în producție.

Un alt obiectiv atins a fost acela de a dezvolta o aplicație ușor de extins. Arhitectura aleasă permite adăugarea de noi funcționalități, precum potențiale îmbunătățiri în zona de comunicare între utilizatori, istoric detaliat al curselor sau notificări avansate.

Pe lângă dezvoltarea tehnică, acest proiect a contribuit semnificativ la consolidarea cunoștințelor privind dezvoltarea aplicațiilor web, interacțiunea cu serviciile cloud, organizarea codului și gestionarea datelor. A reprezentat o ocazie bună de a lucra cu tehnologii moderne într-un scenariu real, apropiat de cerințele industriei.

În concluzie, FastRide nu este doar o lucrare academică, ci și un exemplu de proiect funcțional, care poate servi ca punct de plecare pentru dezvoltări viitoare și care reflectă atât efortul tehnic depus, cât și dorința de a construi aplicații utile, stabile și ușor de folosit.