

Изработка на веб-апликација за споделување на патувања - Carpooling app

Изработил: Кристијан Ристовски, индекс: 196062

1. Вовед

Апликациите од овој тип се ретки во нашата држава, но би биле од огромна корист на граѓаните имајќи ја во предвид состојбата на железниците, автобусите и цената за гориво при приватни патувања. Користејќи го ова како инспирација и успешноста на сличен ваков странски продукт, [Blablacar](#), како финален проект по предметот Напреден веб дизајн, решив да изработам едноставна апликација која би го решила проблемот со патувањата во нашата држава и со помош на оваа апликација да го покажам стекнатото знаење во полето на frontend веб-дизајн.

2. Користени технологии и структура на проектот

Проектот, како скоро секоја модерна веб-апликација, е поделен во два дела, frontend и backend.

Backend делот е на проектот е изработен во **dotNet 8** и е доста едноставен и пригоден на frontend делот за да се прикажат саканите податоци. Како база за податоците ќе користиме локална инстанца на **Microsoft SQL Server** која моделот за табелите ќе го добива директно од backend.

Frontend делот е изработен во Vue 3 и скелетот на целиот проект е изработен со помош на Vite. Како решение за менаџирање со состојба (**state management**) наместо Vuex, во овој проект ќе се користи **Pinia**, како помодерно, побрзо и поедноставено решение, без мутации кои многу го комплицираат целиот проект.

Во следниот дел детално ќе ги анализираме и frontend и backend делот.

2.1. Backend - анализа и опис

Пред да зборуваме за главниот дел од backend делот, треба да анализираме како овој дел од апликацијата се поврзува со база во која ќе ги чуваме нашите податоци. Најпрво, бидејќи работиме во локална околина, на машината мора да се инсталира и покрене инстанца од **Microsoft SQL Server** и истата да се конфигурира. По конфигурацијата, во проектот мора да специфицираме еден **ConnectionString**, прикажан на **Слика 1**, кој му кажува на апликацијата да се поврзе на **localhost\\SQLEXPRESS**, на база **carpoolodb** со дополнителни параметри кои овозможуваат правилно поврзување.

```
"ConnectionStrings": {  
  "DefaultConnection" : "Server=localhost\\SQLEXPRESS;Database=carpoolodb;Trusted_Connection=True; TrustServerCertificate = True;"  
}
```

Слика 1. Специфицирање на Connection string

Следно, ќе го разгледаме моделот со кој опишуваме едно патување. Поради едноставноста на целата апликација, имаме само еден модел кој ја покрива целата сакана функционалност. Комплексноста на моделот може драстично да се зголеми доколку сакаме да

имплементираме корисници, посебни модели за возила, дополнителни проверки и пресметки на податоците. На **Слика 2** можеме да ја видиме структурата на моделот и изгледот на моделот во база.

<pre>namespace carpoolapp_backend.Entities; 4 references public class Ride { 2 references public int Id { get; set; } 0 references public string DriverName { get; set; } 1 reference public string StartLocation { get; set; } 1 reference public string EndLocation { get; set; } 4 references public int AvailableSeats { get; set; } 1 reference public int VehicleSeats { get; set; } 0 references public decimal Price { get; set; } 0 references public DateTime DepartureDate { get; set; } 0 references public RideStatus Status { get; set; } 1 reference public DateTime CreationDate { get; set; } }</pre>										
Id										
DriverName										
StartLocation										
EndLocation										
AvailableSeats										
VehicleSeats										
Price										
DepartureDate										
Status										
CreationDate										
1	John Doe	New York	Boston	2	4	50.00	2024-09-10 00:00:00.0000000	1	2024-09-07 19:58:44.3600000	
2	Jane Smith	Los Angeles	San Diego	3	5	30.00	2024-09-12 00:00:00.0000000	1	2024-09-07 19:58:44.3600000	
3	Mike Ross	Chicago	Detroit	1	4	40.00	2024-09-15 00:00:00.0000000	2	2024-09-07 19:58:44.3600000	
4	Sarah Lee	Houston	Dallas	3	4	60.00	2024-09-18 00:00:00.0000000	3	2024-09-07 19:58:44.3600000	
5	Sam Brown	Miami	Orlando	1	3	20.00	2024-09-20 00:00:00.0000000	1	2024-09-07 19:58:44.3600000	

Слика 2. Модел во код и во база

По дефинирањето на моделот, мораме да го додадеме во податочниот контекст на апликацијата за таа да може да креира соодветно пресликување од моделот специфициран во код во модел читлив за база на податоци. Додавањето на моделот во податочниот контекст се извршува со кодот прикажан на **Слика 3**.

<pre>namespace carpoolapp_backend.Data; 10 references public class DataContext : DbContext { 0 references public DataContext() { } 0 references public DataContext(DbContextOptions<DataContext> options) : base(options) { } 9 references public DbSet<Ride> Rides { get; set; } }</pre>										
---	--	--	--	--	--	--	--	--	--	--

Слика 3. Додавање на модел во податочен контекст

Наредно, ќе го разгледуваме функционалниот дел од backend на апликацијата, односно делот кој нѝ овозможува да земаме и манипулираме податоци од базата. Со контролерите можеме да дефинираме методи кои би овозможиле комуникација со frontend делот и би дозволиле истиот да манипулира со податоците со испраќање на соодветни **requests**. Најпрво го дефинираме контролерот и специфицираме на кој **endpoint** може да се пристапи до него, во овој случај е името на контролерот или endpoint-от ќе биде **/Rides**. Следно го додаваме податочниот контекст во конструкторот на самиот контролер и со тоа ќе пристапуваме до податоците кои ги имаме во базата. Сето ова може да се види на **Слика 4**.

```

namespace carpoolapp_backend.Controllers
{
    [ApiController]
    [Route("[controller]")]
    1 reference
    public class RidesController : Controller {
        private readonly DataContext _context;

        0 references
        public RidesController(DataContext context) { _context = context; }
    }
}

```

Слика 4. Дефинирање на контролер

По дефинирањето на контролерот, ќе го разгледуваме **GET** методот прикажан на **Слика 5**. Со овој метод, интуитивно, ги земаме сите досега креирани патувања од базата на податоци, без користење на никаков филтер, односно ќе ги земеме апсолутно сите записи и ќе ги ставиме во листа.

```

[HttpGet]
1 reference
public async Task<ActionResult<IEnumerable<Ride>>> GetRides()
{
    return await _context.Rides.ToListAsync();
}

```

Слика 5. GET метод

Следно, го разгледуваме **POST** методот за додавање на ново патување, прикажан на **Слика 6**. За овој метод, го изложуваме **/Rides/add** endpoint-от за преку овој endpoint да праќаме податоци од frontend за да ги додадеме во база со помош на дефинираниот метод. По добивањето на податоците за патувањето кое сакаме да го креираме, методот генерира и време на креирање и го додава на патувањето за да имаме преглед кога тоа патување било креирано во база и по успешно додавање во базата, враќаме објект кој го содржи креираното патување.

```

[HttpPost("add")]
0 references
public async Task<ActionResult<Ride>> AddRide(Ride ride)
{
    ride.CreationDate = DateTime.Now;

    _context.Rides.Add(ride);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(GetRides), new { id = ride.Id }, ride);
}

```

Слика 6. ADD метод

Наредно ги ќе ги разгледаме **PUT** методите. **PUT** најчесто се користат за ажурирање на веќе постоечки записи во база, па затоа, во оваа апликација, овој метод ќе служи за ажурирање на бројот на достапни места за седење по патување. Имаме **BookSeat** метод кој го користиме кога сакаме да резервираме место. Како параметар во овој метод го земаме идентификациониот број на патувањето и пробуваме да го најдиме. Доколку не постои враќаме грешка дека не е пронајден бараниот запис. Доколку постои го проверуваме бројот на достапни седишта и доколку е нула, фрламе грешка, а во други случаи го намалуваме бројот, со што кажуваме дека седиштето е резервирано. Аналогно на тоа, откажувањето на седиште работи скоро идентично, освен во проверката на достапните седишта. Тука, доколку достапните седишта се еднакви со бројот на седишта во возилото, тоа значи дека сите места се слободни и нема да се дозволи инкрементација на бројот на седишта, односно дополнително ослободување. Двата методи можат да се видат на **Слика 7** и **Слика 8** соодветно.

```
[HttpPut("{id}/book")]
0 references
public async Task<IActionResult> BookSeat(int id)
{
    var ride = await _context.Rides.FindAsync(id);

    if (ride != null)
    {
        if (ride.AvailableSeats == 0)
        {
            return BadRequest();
        }
        else
        {
            ride.AvailableSeats--;
        }
    }
    else
    {
        return NotFound();
    }

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!RideExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return Ok(ride);
}
```

Слика 7. BookSeat метод

```
[HttpPut("{id}/cancel")]
0 references
public async Task<IActionResult> CancelSeat(int id)
{
    var ride = await _context.Rides.FindAsync(id);

    if (ride != null)
    {
        if (ride.AvailableSeats == ride.VehicleSeats)
        {
            return BadRequest();
        }
        else
        {
            ride.AvailableSeats++;
        }
    }
    else
    {
        return NotFound();
    }

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!RideExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return Ok(ride);
}
```

Слика 8. CancelSeat метод

Последно, во овој контролер ја имаме **DELETE** функционалност. Со методот **DeleteRide**, како параметар земаме идентификационен број на саканото патување и го бришеме од базата на податоци, доколку тоа постои. Методот може да го видиме на **Слика 9**.

```
[HttpDelete("{id}")]
0 references
public async Task<IActionResult> DeleteRide(int id)
{
    var ride = await _context.Rides.FindAsync(id);
    if (ride == null)
    {
        return NotFound();
    }

    _context.Rides.Remove(ride);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Слика 9. Метод за бришење

Дополнително во програмата имаме дефинирано и помошен контролер кој служи за добивање на опциите потребни во паѓачките менија на некои полиња во frontend делот. Контролерот, прикажан на **Слика 10**, го изложува **/api** endpoint преку кој ќе пристапиме до саканите методи во овој контролер. Исто како и претходно, го додаваме податочниот контекст во инстанцата. Дефинираме **GET** метода на **/api/locations** endpoint со која ги земаме сите почетни и крајни локации и уникатните вредности ги ставаме во една листа, за да добиеме комплетна резултантна листа од сите локации во сите поединечни патувања во што ги имаме во базата на податоци.

```
namespace carpoolapp_backend.Controllers
{
    [ApiController]
    [Route("api")]
    1 reference
    public class UtilsController : Controller
    {
        private readonly DataContext _context;

        0 references
        public UtilsController(DataContext context) { _context = context; }

        [HttpGet("locations")]
        0 references
        public async Task<ActionResult<IEnumerable<string>>> GetLocationOptions()
        {
            var startLocations = await _context.Rides
                .Select(r => r.StartLocation)
                .Distinct()
                .ToListAsync();

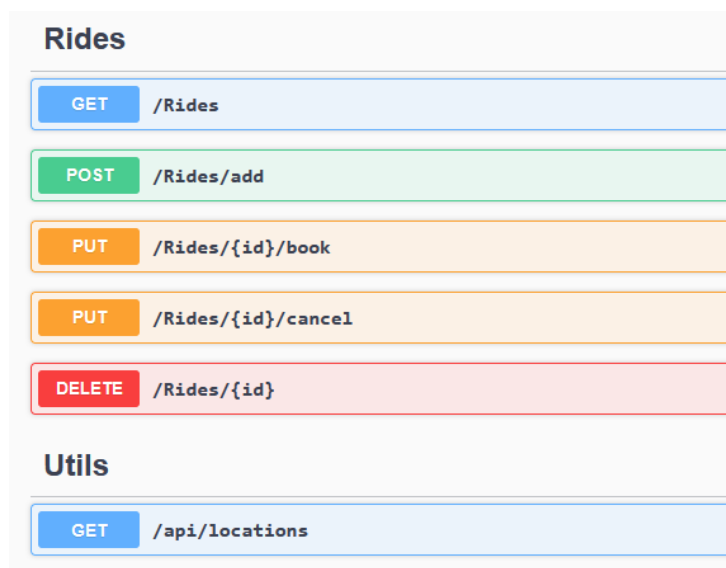
            var endLocations = await _context.Rides
                .Select(r => r.EndLocation)
                .Distinct()
                .ToListAsync();

            var allLocations = startLocations.Concat(endLocations)
                .Distinct()
                .OrderBy(l => l)
                .ToList();

            return allLocations;
        }
    }
}
```

Слика 10. Помошен контролер за локации

Со тоа го опфативме целиот Backend дел од апликацијата, и на **Слика 11** можеме да ги видиме сите изложени endpoints од соодветните контролери од **Swagger** документацијата.

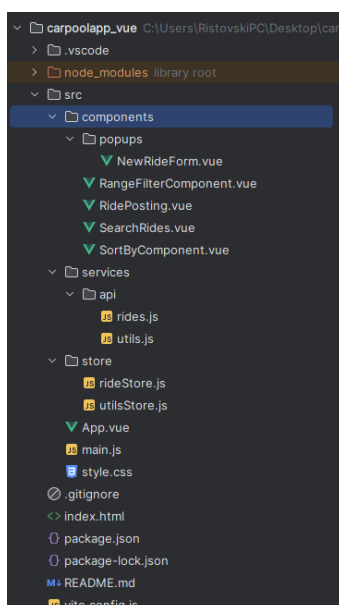


Слика 11. Изложени endpoints

2.2. Frontend - анализа и опис

Frontend делот е далеку по комплексен и екстензивен од претходно опишаниот backend. Големiot дел од бизнис логиката, филтрирања, сортирања и слично, се одвиваат во frontend. Во пракса, ова не се практикува бидејќи за големи проекти, големиот број на податоци што треба да се преработат, најчесто го кочат целиот веб-пребарувач и затоа сета таа потешка логика е префрлена на backend. Но, за овој проект, малиот број на податоци и фокусот на предметот на frontend технологиите, целата логика е префрлена во овој дел. Дополнително, за изгледот на апликацијата, користена е библиотеката **Vuetify** која содржи веќе стилизирани компоненти.

На **Слика 12** е претставена структурата на проектот и по неа ќе ја вршиме анализата на кодот. Најпрво ќе зборуваме за помошните функции во **services** датотеката и **state management** делот во **store** датотеката.



Слика 12. Структура на frontend проект

2.2.1. Помошни фунцкии - Services

Во сервисите имаме само функции кои комуницираат со backend на апликацијата. Тие функции се сместени во **api** папка во соодветни **JavaScript** датотеки. Во **rides.js** се наоѓаат помошни функции за повикување на **endpoints** од **Rides** контролерот, а во **utils.js** се наоѓаат повиците до **Utils** контролерот. Како што може да се види на сликите 13 и 14 подолу, ја користиме **axios** библиотеката за реализирање на повиците до backend до соодветните endpoints.

```
import axios from "axios";

export async function getRidesHelper() { Show usages  Kristijan
  return await axios.get(url: 'https://localhost:7125/Rides');
}

export async function createRideHelper(rideData) { Show usages  Kristijan
  return await axios.post(url: 'https://localhost:7125/Rides/add', data: {
    driverName: rideData.driverName,
    startLocation: rideData.startLocation,
    endLocation: rideData.endLocation,
    availableSeats: rideData.availableSeats,
    vehicleSeats: rideData.vehicleSeats,
    price: parseFloat(rideData.price),
    departureDate: rideData.departureDate.toISOString(),
  });
}

export async function bookSeat(id) { Show usages  Kristijan
  return await axios.put(url: 'https://localhost:7125/Rides/${id}/book');
}

export async function cancelSeat(id) { Show usages  Kristijan
  return await axios.put(url: 'https://localhost:7125/Rides/${id}/cancel');
}

export async function deleteRide(id) { Show usages  Kristijan
  return await axios.delete(url: 'https://localhost:7125/Rides/${id}');
}
```

Слика 13. Содржина на rides.js

```
import axios from "axios";

export async function fetchLocationOptions() { Show usages  Kristijan
  return await axios.get(url: 'https://localhost:7125/api/locations');
}
```

Слика 14. Содржина на utils.js

2.2.2. State management - Pinia stores

Во папката **stores**, ги имаме двете складишта на нашата апликација, едната за патувањата, **rideStore.js** и едната за помошните податоци, **utilsStore.js**.

```
import {defineStore} from "pinia";

import {fetchLocationOptions} from "../services/api/utils.js";

export const useLocationsStore = defineStore({id: 'Locations', options: { Show usages
  state: () => ({
    locations: []
  }),

  actions: {
    async getLocationOptions() {
      if (this.locations.length){
        return this.locations;
      }
      const result = await fetchLocationOptions()
      this.locations = result.data;

      return this.locations;
    }
  }
});
```

Слика 14. Содржина на utilsStore.js

Во **utilsStore.js** ја чуваме состојбата на локациите и сакаме да персистира низ целата апликација. Правиме повик до backend да ги добиеме податоците и ги чуваме во глобално достапна променлива за состојба, па кога и да имаме потреба од пристап до овие податоци, од било која компонента, можеме да пристапиме без повторно да праќаме повик до backend.

Во **rideStore.js** ја чуваме состојбата на патувањата и дополнително правиме сортирање и филтрирање. Во **state** делот ги чуваме параметрите за сортирање, интервалот на максимална и минимална цена на патувањата, иницијалните патувања без никаква манипулација и иницијалниот интервал на максимална и минимална цена и **rides** листата која всушност е конечната листа по сите пресметки. Оваа листа ќе се користи низ апликацијата. Во **actions** делот ги правиме сите пресметки. Методот **getAllRides** го користиме за да ги земиме податоците од backend со користење на функцијата од **api**. Резултатот го ставаме во **initialRides** и овој резултат го користиме за да се навратиме доколку има потреба на иницијалната листа на патувања, а со помош на оваа листа го пресметуваме иницијалниот интервал на цени од патувањата. Правиме плитка копија на иницијалните патувања во **rides** променливата и продолжуваме со сортирање на истата со поставените параметри за сортирање. Имаме и две функции за ажурирање на параметрите за сортирање и интервалот на цени. Методот за сортирање **sort** ги подредува елементите во листата во зависност од тоа што имаме како параметар за сортирање, а **rangeFilter** методот проверува кој елемент од листата е во интервалот на цени кој го имаме моментално.

```
import {defineStore} from 'pinia'
import {getRidesHelper} from '../services/api/rides.js';

export const useRidesStore = defineStore({ id: 'Rides', Show usages  Kristijan
  options: {
    state: () => ({
      initialRides: [],
      initialPriceRange: {
        lower: null,
        upper: null
      },
      rides: [],

      sortBy: null,
      ascending: false,

      currentPriceRange: {
        lower: null,
        upper: null,
      }
    })
  },
```

```
  actions: {
    async getAllRides() {
      try {
        const result = await getRidesHelper();
        if (result) {
          this.initialRides = result.data;
          this.calculateInitialLowerUpper();

          this.rides = [...result.data]; // shallow copy
          await this.sortRides();
        }
      } catch (e) {
        console.log(e);
      }
    },

    calculateInitialLowerUpper() {
      if (this.initialRides.length) {
        this.initialPriceRange.lower = Math.min(...this.initialRides.map(r => r.price));
        this.initialPriceRange.upper = Math.max(...this.initialRides.map(r => r.price));
      }
    },

    async updateSortParams(sortBy, ascending) {
      this.sortBy = sortBy;
      this.ascending = ascending;

      await this.sortRides();
    },

    updatePriceRangeParams(lowerValue, upperValue) {
      this.currentPriceRange.lower = lowerValue;
      this.currentPriceRange.upper = upperValue;

      this.rangeFilter();
    },
```

```
  },

  async sortRides() {
    if (!this.sortBy) {
      this.rides = [...this.initialRides]; // shallow copy to reset
      return;
    }

    this.rides.sort((a, b) => {
      let valueA = a[this.sortBy];
      let valueB = b[this.sortBy];

      if (this.sortBy === 'departureDate' || this.sortBy === 'creationDate') {
        valueA = new Date(valueA);
        valueB = new Date(valueB);
      }

      return this.ascending ? valueA - valueB : valueB - valueA;
    });
  },

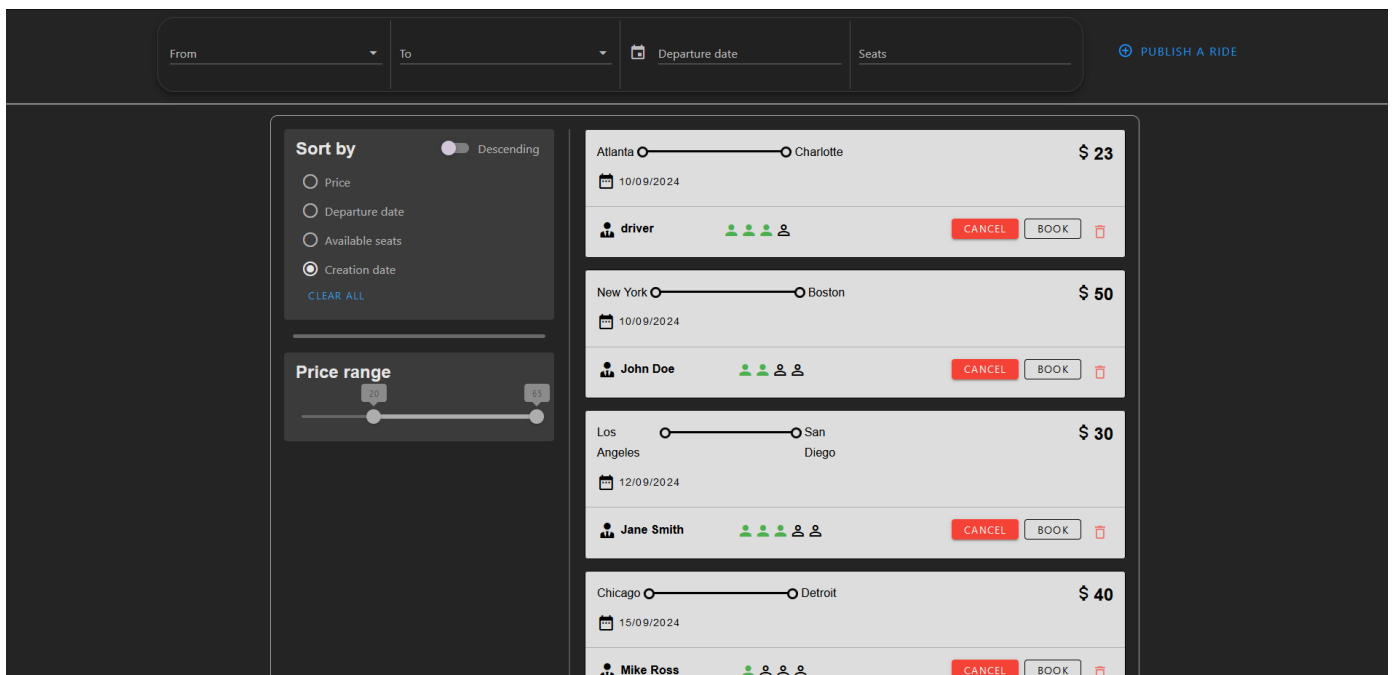
  rangeFilter() {
    this.rides = [...this.initialRides
      .filter(r => r.price >= this.currentPriceRange.lower && r.price <= this.currentPriceRange.upper)];
  }
});
```

Слика 15. Содржина на rideStore.js

2.2.3. Компоненти - изглед и функционалност

- App.vue

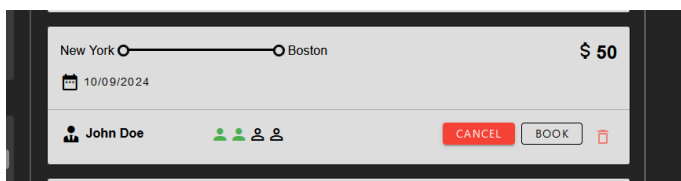
Оваа компонента е главната компонента во која се сместени сите останати компоненти, односно оваа компонента е родител на сите понатаму опишани компоненти. Во **template** делот од компонентата е сместен делот од апликацијата што се покажува на екран, односно HTML, други компоненти и слично. Како што може да се види на **Слика 16** имаме повеќе компоненти кои ќе ги опишуваме поединечно. Во оваа компонента имаме логика за земање на податоците од **store**, прикажување на патувањата, прикажување на компонента за **loading** додека се земаат истите податоци, логика за филтрирање на податоците, односно во оваа компонента се специфицираат параметрите со кои се филтрира во позадина.



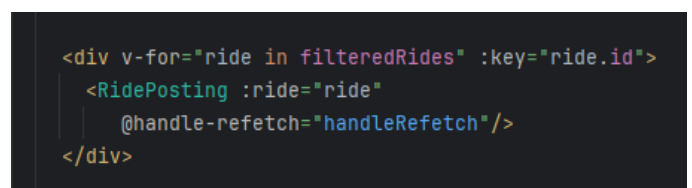
Слика 16. Изглед на апликацијата (App.vue)

- RidePosting.vue

Компонентата со која ги прикажуваме патувањата е претставена на **Слика 17**, неа ја рендерираме во **App.vue** со извадокот прикажан на **Слика 18**. Како што може да се види, на компонентата како **prop** му го праќаме **ride** објектот кој ги содржи сите податоци за тоа патување и компонентата соодветно се справува и ги прикажува тие информации. Слободните седишта се визуелизирани со помош на икони кои слободните места ги покажуваат со зелена боја, а зафатените како обезбоени икони. Имаме и копчиња за резервирање и откажување место, како и копче за бришење на патување. Сите копчиња ги користат соодветните методи за повикување на методите од backend.



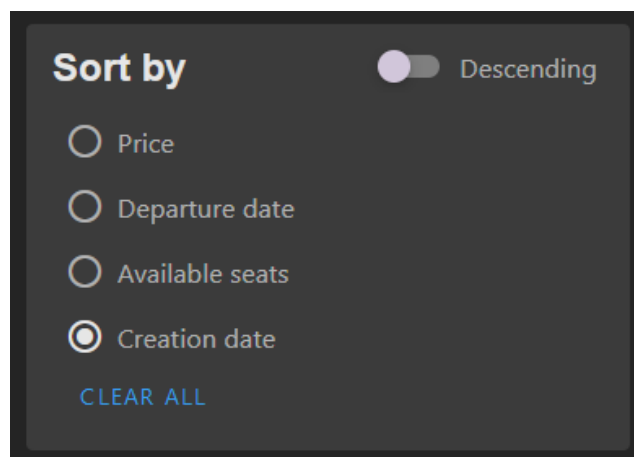
Слика 17. Изглед на RidePosting.vue



Слика 18. Извадок од App.vue за рендерирање на компонентата

- [SortByComponent.vue](#)

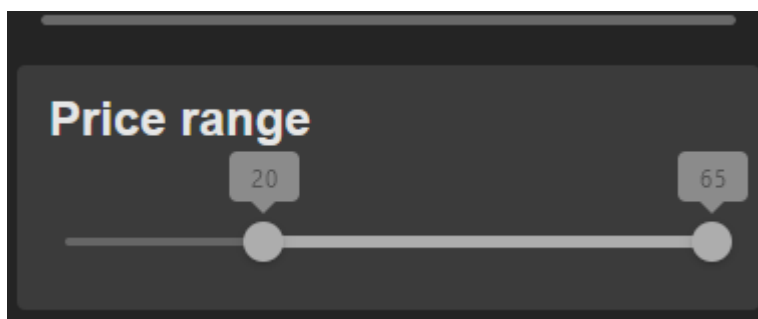
Со оваа компонента ги поставуваме параметрите за сортирање на листата на патувања. По секоја промена на параметрите, компонентата прави соодветен повик до функцијата за ажурирање на параметрите во **rideStore** и понатамошната функционалност му ја остава неа. Повиците до функциите се одвиваат со помош на **watch**, односно Vue ја следи состојбата на одреден параметар во компонентата и доколку се промени, ивршува некоја функција, зададена во телото на watcher-от. Целта на оваа компонента е само тоа, одбирањето на параметрите за сортирање.



Слика 19. Изглед на SortByComponent.vue

- [RangeFilterComponent.vue](#)

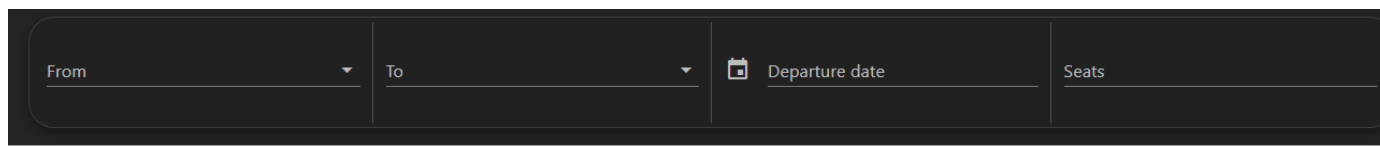
Оваа компонента користи дво-краен лизгач за да го поставиме интервалот на цени со кој сакаме да ги филтрираме патувањата. Во оваа компонента како **props** од главната компонента ги праќаме иницијалните вредности на минимумот и максимумот и ги чуваме за да го ресетираме лизгачот кога посакаме. Секое понатамошно менување на позицијата на лизгачот, покренува метод за ажурирање на параметрите во **ridesStore** за да ги изфилтрира патувањата кои не се во бараниот интервал.



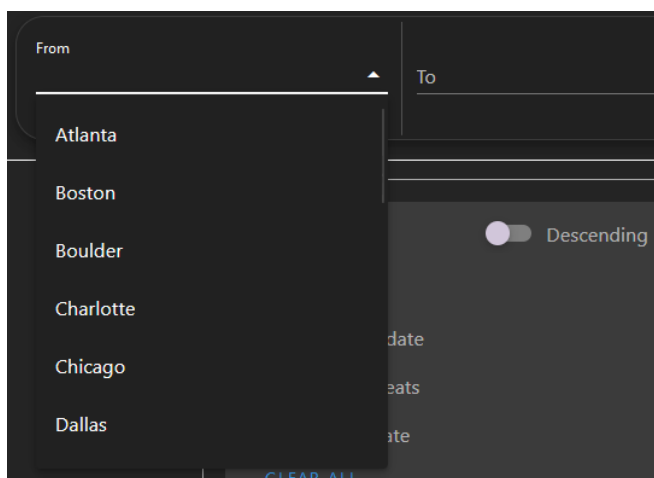
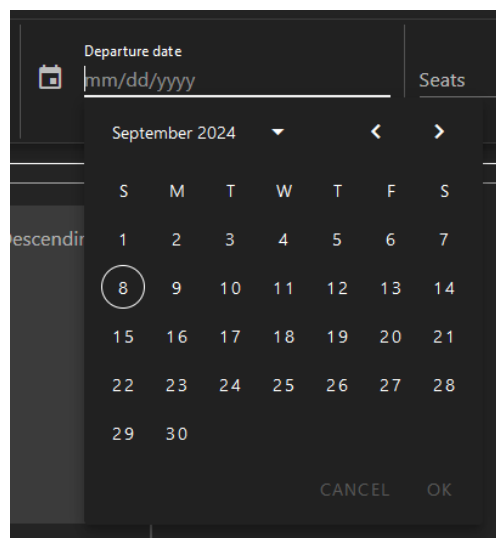
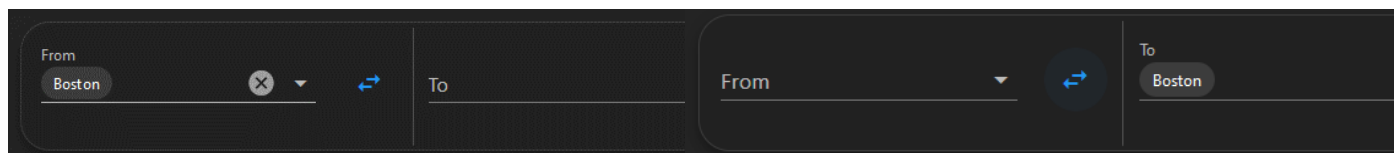
Слика 20. Изглед на RangeFilterComponent.vue

- [SearchRides.vue](#)

Една од покомплицираните компоненти во овој проект, **SearchRides** ја користиме за пребарување на патувањата по одредени параметри. Првите две полиња за одбирање на почетна и крајна локација, имаат **autocomplete** функционалност која пребарува по листа од локации, а може и да се одбере од паѓачко мени со истите локации низ кои пребарува. Исто така има функционалност за промена на вредностите од едно во друго поле, прикажано подолу на **Слика 22**. Полето за внесување датум прикажува календар во кој може да се одбере некоја дата во **mm/dd/YYYY** формат. Форматот не може да се промени поради тоа што компонентата е сеуште некомплетна и во фаза на тестирање од страна на Vuetify, но ова не прави голем проблем низ апликацијата бидејќи сепак овој датум се конвертира во **JavaScript Date** објект кој има стандарден изглед. Во компонентата без да кликнеме на копче за пребарување, нашите параметри ќе се проследат низ апликацијата. За таа цел користиме **watcher** за да препознае промени во полињата и доколку има некаква промена, повикува **debounce** метод кој всушност претставува функција која се извршува по одредено време. Оваа функционалност ја добиваме од библиотеката **lodash**.

A dark-themed form with four input fields: 'From', 'To', 'Departure date' (with a calendar icon), and 'Seats'. Each field has a dropdown arrow on its right side.

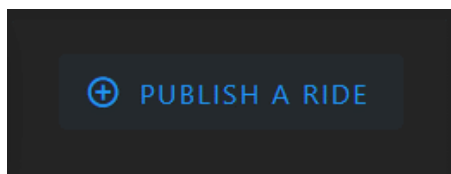
Слика 21. Изглед на SearchRides.vue

A close-up of the 'From' field's dropdown menu. It lists several cities: Atlanta, Boston, Boulder, Charlotte, Chicago, and Dallas. To the right of the list is a 'Descending' toggle switch. At the bottom of the menu is a 'CLEAR ALL' button.A close-up of the 'Departure date' field's calendar. It shows the month of September 2024. The days of the week are listed at the top: S, M, T, W, T, F, S. The dates 1 through 30 are arranged in a grid. The date 8 is circled. At the bottom are 'CANCEL' and 'OK' buttons.A close-up of the 'From' and 'To' fields. The 'From' field contains 'Boston'. There is a swap button (two arrows) between the 'From' and 'To' fields. The 'To' field is empty.

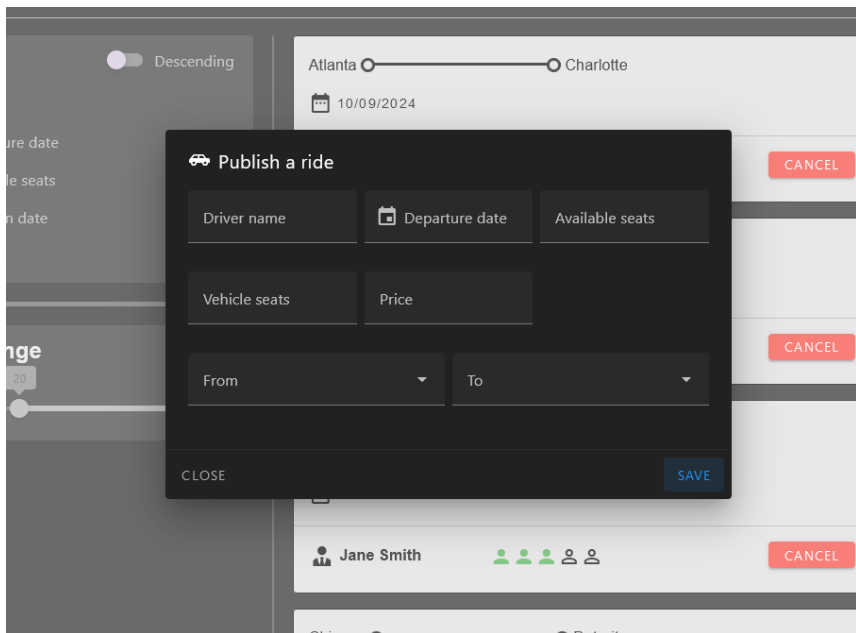
Слика 22. Функционалности на SearchRides.vue

- [NewRidePosting.vue](#)

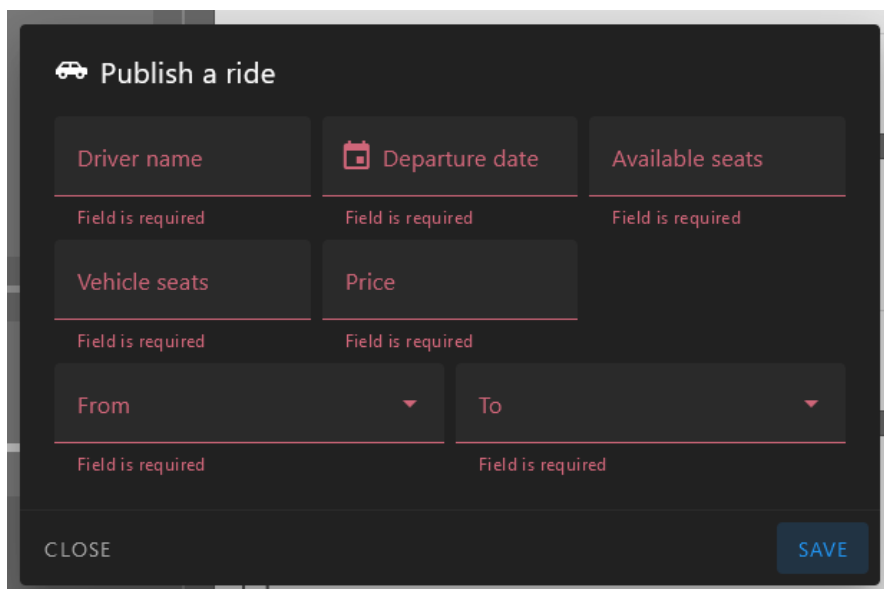
Како што ни кажува името на компонентата, оваа компонента се користи за додавање на ново патување со помош на рорир форма. Со кликање на копчето претставено на **Слика 23**, ја активираме формата и излегува како рорир како што е прикажано на **Слика 24**. Формата во неа има правила кои се доделуваат поединечно на секое поле и доколку едно од тие правила е неисполнето, формата нема да дозволи да се пратат податоците до backend. Но, доколку таа е валидна и сите податоци се во склад со поставените правила, во компонентата се повикува методот за праќање на ново патување до backend и истата се затвора, бришејќи ги досега внесените податоци во полињата.



Слика 23. Копче со кое ја уключуваме рорир формата



Слика 24. Изглед на рорир формата



Слика 25. Изглед на формата доколку не е валидна

3. Можности за скалирање и дополнителни функционалности

Како и сите модерни веб-апликации и оваа апликација има огромен потенцијал за скалирање. Покрај тоа што може да се подобри во изглед и да се подобри корисничкото искуство, туку може и да се додадат огромен број на нови функционалности, како најава на корисници, оценување на возачи, дополнителни валидации на формите, поробустен backend, миграција на голем дел од логиката во backend итн. Една функционалност која беше иницијална идеја да се воведи и во овој проект беше користењето на екстерно API, пример Google Maps API, за да се добие листа од градови и места кои потоа ќе може да се изберат како опции и со гео локација да се пресмета времето за патување и слично. Но тоа остана само идеја поради тоа што скоро сите API се со ограничување и клучните функционалности се недостапни без плаќање релативно големи суми, па затоа таа идеја остана нереализирана.