Mykyta DERMENZHI[1], Tetiana TERESHCHENKO[2]

Supervisor: Oleksii FRAZE-FRAZENKO[3]

OPRACOWANIE SYSTEMU ZARZĄDZANIA I ŚLEDZENIA NAWIGACJI DLA SYSTEMU MOTORYZACYJNEGO W CZASIE RZECZYWISTYM

Streszczenie: Podstawowym tematem tej pracy jest wyzwanie dla systemu o dużym obciążeniu z danymi liczbowymi i ogromnymi danymi, tworzącego duży projekt z możliwością późniejszej rozbudowy w przyszłości i umożliwiającym dostarczanie nowych funkcji bez zmiany już istniejących.
Niektóre z celów to: dostarczanie fragmentów danych geograficznych w celu uniknięcia niepotrzebnego obciążenia maszyn klienckich, aktualizowanie obiektów motoryzacyjnych w czasie rzeczywistym oraz przybliżone obliczanie odległości i czasów, szacowanie tras, które powinny odpowiadać realistycznym wynikom, automatyczne powiadomienia dla użytkowników systemu. Cała architektura zostanie zrealizowana w postaci struktury mikroserwisów.
System będzie oparty na jednej z wersji platformy .NET Core i będzie korzystał z systemu Micro-ORM takiego jak Dapper. "Wanilowy" JavaScript zapewni możliwość największego wsparcia na różnych platformach, a niektóre wtyczki z Leaflet zapewnią najbardziej przyjazny interfejs użytkownika dla dostawcy OSM (Open Street Map).

Słowa kluczowe: .NET Core, ASP, JSON, Dapper, mikroserwisy, Leaflet

DEVELOPMENT OF A MANAGEMENT AND NAVIGATION TRACKING SYSTEM FOR THE AUTOMOTIVE SYSTEM IN REAL TIME

Summary: Primary topic of this work is a challenge for high-load system with a numeric and massive data, creating a large project with following extending in the future and allowing for providing new features without touching existing ones. Some of aims are: providing chunks with geo-data for avoiding unnecessary load on client machines, real-time updating for

[1] Student of Computer Science, Management and Administration Faculty, Odesa State Environmental University, nikita.dermenzhi@gmail.com
[2] Engineering Science Ph.D., Odesa State Environmental University, associate professor at the department of information technology, tereshchenko.odessa@gmail.com
[3] Engineering Science Ph.D., Odesa State Environmental University, associate professor at the department of information technology, frazenko@gmail.com

automotive objects and providing the approximate calculating of distances and timings, estimates for routes what should match with realistic results, automatic notifications for users of the system. The whole architecture will be implemented in a way of microservices structure. The system is going to be based on the one of the versions on .NET Core platform and will be using Micro-ORM system such as a Dapper. "Vanilla" JavaScript will provide the most possible support on different platforms and some plugins from Leaflet will allow provide the most friendly UI for OSM (Open Street Map) provider.

Keywords: .NET Core, ASP, JSON, Dapper, Microservices, Leaflet

1. Introduction

Nowadays, every single operator of high-load system has at least once a thought about how it can be simplified or even better upgraded. The most middle-range companies have some problems with structuring and investigation and deeping in data. For example, a lot of problems in underdeveloped countries are caused by bureaucracy in the core of business.

Globally, a subject area of the project is a navigation system with built-in real-time management and notification coroutines.

So, for the most productive implementation and design system should be set up all services and layers of it. Start point for the beginning will be layering process, which will produce the parts and define their dedications.

The best way for successful brainstorm is a beginning from diagramming the model and structuring it (fig. 1):

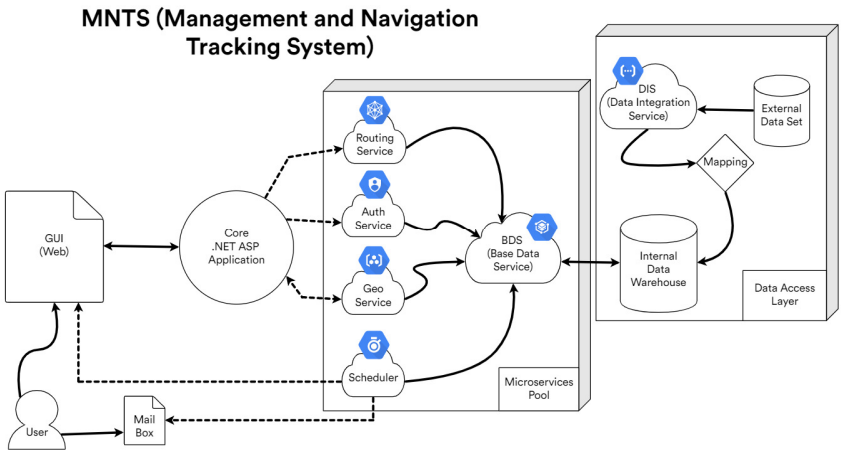

Figure 1. Layering the system

As from diagram can be seen – complex part will be Microservices Pool and the most important things will be happened exactly there [1]. They will have shared parent Base Data Service, further BDS. BDS is the common component with connection to Data Access Layer (DAL) and some of necessary plug-ins: mapper, service data, etc. Internal Data Warehouse is an abstract layer which can be switched between

providers, the only contract it should have is entities are existed in the core application and correct procedures' names.
The system is going to be working with integration data which can be provided from the consumer storage. External Data Set is the simple set of a data what should be processed by Data Integration Service (DIS) and move to mapping workflow where the external data turn into internal model. Initial population is a test data set and it will fit the integration workflow. In future the integration could be extended with a scheduler what might allow the system synchronize two unconnected systems and set them up-to-date.
Auth Service (AS) is a new generation of token system which is called JWT [2] (JSON Web Token). This service allows identify user using only a string and allows system to seal data right into this string. This feature is a progressive step which provides using some meta data from user's requests. The common view of token can be seen below:

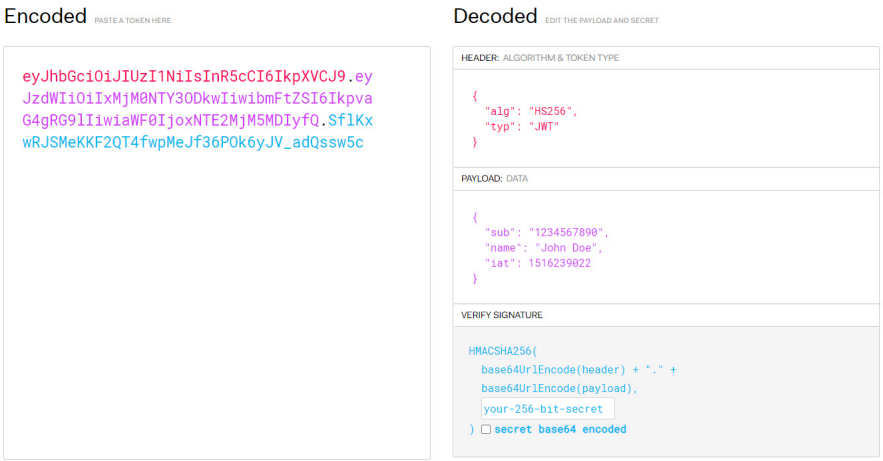

Figure 2. JWT demonstration

Forming a payload or algorithm will change the whole string even if it's only one symbol change. So the full hash will show to application the real and secureness action for each of requests. The demo site from where the information and testing have been prepared: https://jwt.io/.
Geo Service (GS) is a heart of the project and the main part. This service will consist of many coroutines and will work with data set. This means that GS influences on data directly and it will work with data writing and reading it from a warehouse. Also, it should build linked lists and connect some additional data for locations and objects. In the pool of using in this service can be enumerated some of the global mechanisms: Bus Service, Driver Management Service, Trip Management Service, etc.

Before start the project one of the most important parts is choosing the good way for calculation and routing trips. Definitely, can be written the own module which can cover the useful cases, but today in the world humanity has a lot of great and good solutions and tones of providers of it.
So, it was decided to work with one of the most accurate and widely-spreaded systems – TomTom [3]. **Routing Service** (RS) will be based and configured using endpoints

of API of a Dutch company. And, can be noticed that even this service is external, but for decreasing requests there will be used internal data saving functions. For example, if some route has the same locations the system will use saved data instead of creating a request outside. This will save money to customer and also will parse and populate useful data for further developments.
Scheduler – this will be a singleton coroutine which can be as standalone service or built-in part of developed system. Basically, that will be a timer with a queue of tasks, which are being executed once the time comes. The most important task of existed should be the notification system for users and mailbox updates, but further it might be extended by a specific requirements from each of consumers.

2. Structure and design

The next of step of lifecycle process should be structuring and defining every independent layer [4]. Essential thing for every information system is a data warehouse, in this individual case was decided to use lightweight and fast bundle: SQL Server and Dapper. This bunch allows to make a simple actions to database through a general interface of every connection, what means that we can swap one of node (Database) and replace if it's needed. But the only ought database should have is the full roster of entities predefined on designing step.
For initial database setup should be used scripts in the order of their names (fig. 3), e.g.: 1.sql, 2.sql, etc. Each of scripts do their obligations and set their area up.

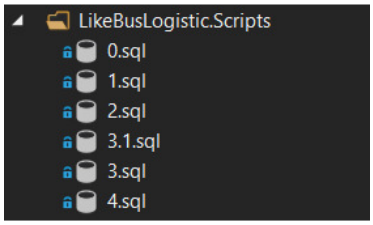

Figure 3. Scripts hierarchy

The best way for understanding what area is covered by a script is to look at it. Before doing that below is provided short description of them:
0.sql – creating all tables and initiating execution of some system functions.
1.sql – additional script for temp tables and integration processes if needed.
2.sql – procedures and custom functions initializing.
3.1.sql – test data definition.
3.sql – production data definition if existed.
4.sql – auto-mapping script for proceeding models to Dapper ORM.
For short demo, would be a great point to show some part:

```
if not exists (select 1
               from sys.tables t
               where t.name='Role'
               and t.schema_id = schema_id('dbo'))
  create table dbo.Role
```

```
  (
    Id int not null primary key identity,
    Name nvarchar(20) not null unique,
    DateCreated datetime not null default(dbo.getUAdate()),
    DateModified datetime not null default(dbo.getUAdate()),
    IsDeleted bit not null default(0)

    constraint UQ_dbo_Role_Name unique (Name)
  );
go
```

Here can be seen an idempotent script of creating table from script 0.sql. Idempotent means that user can start running it infinite amount of times, but still won't get any issues with the data model. Also, for short demo purposes below is a part of a script from procedures sql, 2.sql:

```
if object_id(N'dbo.GetUserAccountById') is null
  exec('create procedure dbo.GetUserAccountById as set
nocount on;');
go

-- Example    : exec dbo.GetUserAccountById 'a', 'a'
-- Author     : Nikita Dermenzhi
-- Date       : 25/07/2019
-- Description: —

alter procedure dbo.GetUserAccountById(@id as int)
as
begin
  select a.Id         as AccountId
       , r.Id         as RoleId
       , r.Name       as RoleName
       , u.Id         as UserId
       , u.FirstName  as FirstName
       , u.LastName   as LastName
       , u.MiddleName as MiddleName
       , u.Phone      as Phone
       , u.Email      as Email
    from Account a
    join [Role] r on a.RoleId = r.Id
    join [User] u on a.UserId = u.Id
    where 1=1
      and a.Id = @id
      and a.isDeleted = 0
      and r.isDeleted = 0
      and u.isDeleted = 0

end;
go
```

From this stored procedure user can get information from several tables are joined by foreign keys (FK) on their primary keys (PK) using only one parameter – user identifier. This procedure will be used in **AS** for getting data of request and authenticate user's action.

Next step is an external services [5] that can be used not only in this project and for the outsource using purposes they were transferred there (fig. 4):

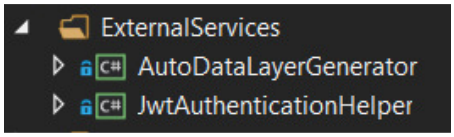

Figure 4. External services

AutoDataLayerGenerator – is a specific utility was designed for generating a real C# models from a database model. That allows to forget about rewriting the same column names to the properties and always get up-to-date model just by running this tool one time.

JwtAuthenticationHelper – is an abstract wrapper of standard JWT authentication provided by Microsoft team, but with some changes with sealing metadata to a token and decrypting data in middleware before request will start acting, all of this features were discussed above in a chapter 1.

Valuable period in designing is a creation of business logic layer (BLL) and the main requirement to this layer was a maximum possible split up on a small pieces for easy develop and supportive end in future. After thinking over the subject area was decided to split up the **BLL** on 11 parts (fig. 5.):

- AccountManagementService;
- AnonymousServiceFactory – factory for producing anonymous services;
- BaseService – a parent for each of the service;
- BusManagementService;
- DriverManagementService;
- GeolocationService;
- LookupManagementService – a dictionary, can be used for multi-language;
- RouteManagementService;
- ScheduleManagementService;
- TomTom – a service to external API for calculating and prediction routes;
- TripManagementService.

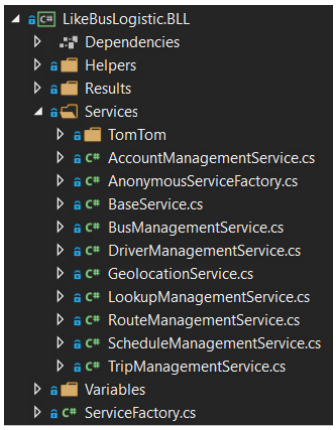

Figure 5. Services infrastructure

For better comprehension lower is shown on example of BaseService the code basis of these entities:

```
public abstract class BaseService : IDisposable
{
    private string _connectionString
    public const string GeneralSuccessMessage = "Успешно
выполнено!";
    public const string GeneralErrorMessage = "Произошла
ошибка!"
    private IMapper _mapper;
    protected IMapper Mapper => _mapper ?? (_mapper = new
ServiceMapperExtension().Mapper);
    protected UnitOfWork UnitOfWork { get; set; }
    protected IDbConnection Connection => new
SqlConnection(_connectionString)
    public int? AccountId { get; set; }
    public bool Anonymous => !AccountId.HasValue &&
AccountUserRole == null;
    public AccountUserRoleVM AccountUserRole =>
    AccountId.HasValue
    ?
Mapper.Map<AccountUserRoleVM>(UnitOfWork.StoredProcedureDao.
GetUserAccountById(AccountId.Value))
    : null;
    public RoleName RoleName
    {
        get
        {
            RoleName role;
            if (AccountUserRole?.RoleName ==
RoleName.Administrator.ToString())
            {
                role = RoleName.Administrator;
            }
            else if (AccountUserRole?.RoleName ==
RoleName.Moderator.ToString())
            {
                role = RoleName.Moderator;
            }
            else if (AccountUserRole?.RoleName ==
RoleName.Operator.ToString())
            {
                role = RoleName.Operator;
            }
            else
            {
                role = RoleName.Unknown;
            }
            return role;
        }

    public BaseService(string connection)
    {
        _connectionString = connection;
        UnitOfWork = new UnitOfWork(Connection);
```

```
        public void Dispose()
        {
            UnitOfWork.Dispose();
        }
    }
```

As decided in introduction the BaseService should have such properties as Mapper, Connection, Account and Role, etc. These fields allows in inherited classes get extra info for changing the behavior or for fluenting on basic methods.

Here, in this module was used the most popular techniques: facade, unit of work and repository patterns [6]. The project was designed on principles from the book "Gang of Four" has been written by programmers from 1994.

Facade [7] is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Repositories are classes or components that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer (fig. 7).

Unit of work [8] maintains a list of objects affected by a business transaction and coordinates the writing out of changes. ("Patterns of Enterprise Application Architecture" by Martin Fowler).

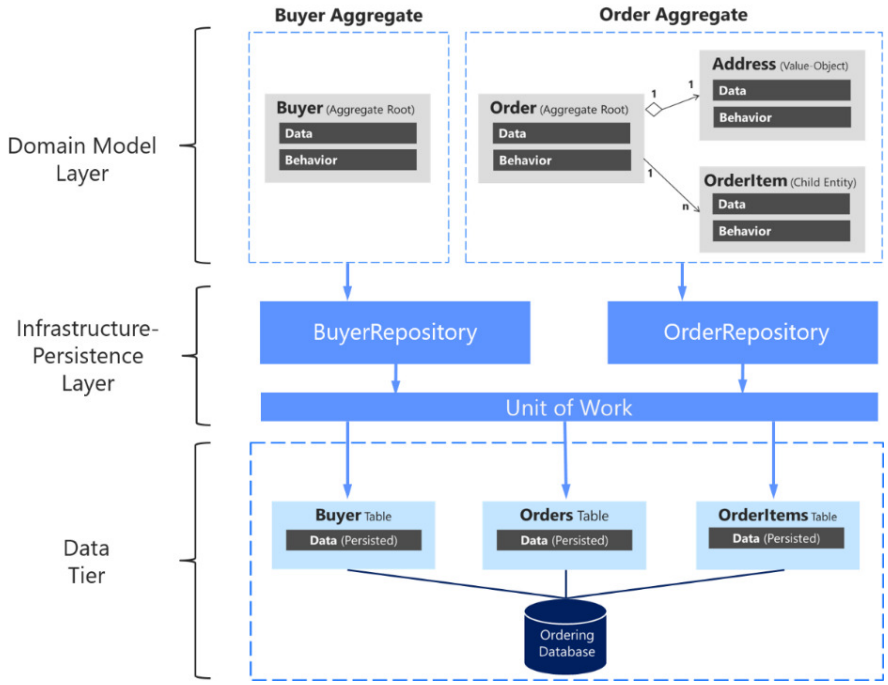

Figure 6. The relationship between repositories, aggregates, database tables

One more pattern was involved into building DAL, Data Access Object pattern or DAO. The Data Access Object (DAO) pattern is a structural pattern that allows to isolate the application/business layer from the persistence layer (usually a relational database, but it could be any other persistence mechanism) using an abstract API.
For catching the way how BaseService gets data from database and change it, need to be presented a data access layer (DAL) that a sub-layer for interacting with data warehouse (fig. 7). Lower is presented hierarchy of data access objects which will be a source for an information in core application.

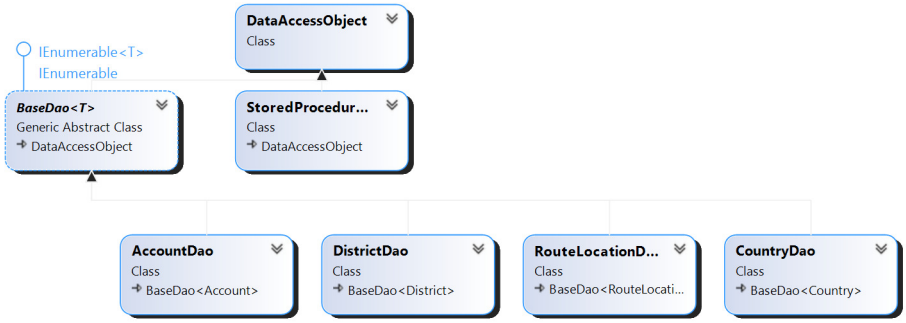

Figure 7. A part of DAOs hierarchy

For full view class diagram, check the link:
https://raw.githubusercontent.com/bladehero/LikeBusLogistic/master/LikeBusLogistic.DAL/ClassDiagram.png

One of serious processes in application is mapping, this operation can be between each layer and the problem it solve is dependency inversion (DI) between abstract wrappers (fig. 8). So, the point of DI is to make easy developing each of system and don't think about models as a key objects. In the end each of layers will be using their own models, which usually called ViewModel (VM) and after combining and constructing/compiling all of parts together the only need the architecture will have is just create a mapping rules for connecting every VM. VM is a domain model, but domain for a concrete lay.

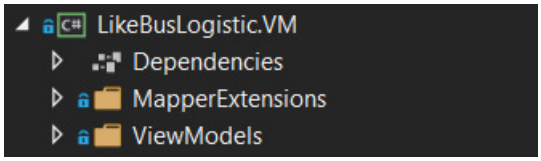

Figure 8. Connecting abstract layers in one module – VM

DI [9] – is one of principles from a great strategy SOLID. So this project will also implement all advantages from this monumental work and collect all best practices together (fig. 9).
With the addition of an abstract layer, both high- and lower-level layers reduce the traditional dependencies from top to bottom. Nevertheless, the "inversion" concept does not mean that lower-level layers depend on higher-level layers. Both layers should depend on abstractions that draw the behavior needed by higher-level layers.

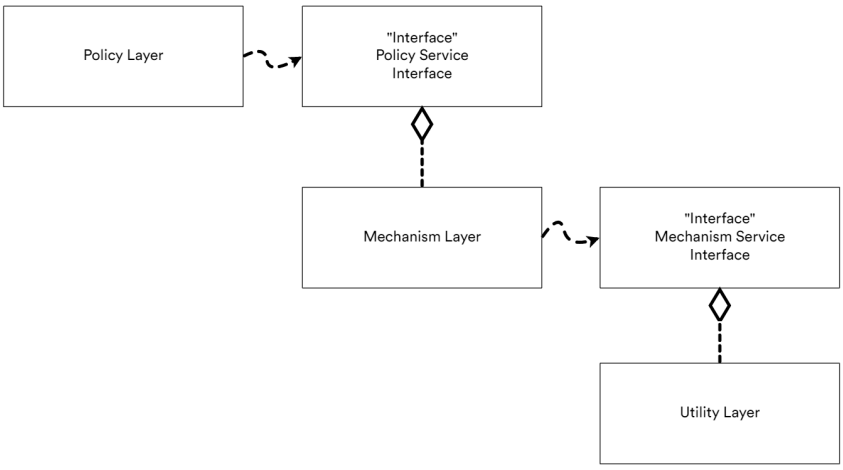

Figure 9. Dependency Inversion Pattern

As example, will be taken ServiceMapperExtension, which can show how it works and for what purposes it was introduced:

```
public class ServiceMapperExtension : BaseMapperExtension
{
    public ServiceMapperExtension()
    {
        _config = new MapperConfiguration(cfg =>
        {
            #region Account Management Service
            cfg.CreateMap<Account, AccountVM>();
            cfg.CreateMap<AccountVM, Account>();
            cfg.CreateMap<Role, RoleVM>();
            cfg.CreateMap<RoleVM, Role>();
            cfg.CreateMap<User, UserVM>();
            cfg.CreateMap<UserVM, User>();
            cfg.CreateMap<GetUserAccountById_Result,
AccountUserRoleVM>();

cfg.CreateMap<GetUserAccountByCredentials_Result,
AccountUserRoleVM>();
            cfg.CreateMap<AccountUserRoleVM,
GetUserAccountById_Result>();
            cfg.CreateMap<AccountUserRoleVM,
GetUserAccountByCredentials_Result>();
            #endregion
            #region Bus Management Service
            #region Driver Management Service
            #region Geolocation Service
            #region Route Management Service
            #region Schedule Management Service
            #region Trip Management Service
            #region Lookup Management Service
        });
    }}
```

As could be seemed this is extra work, but further Mapping will present itself and all questions will disappear. But as was said this allow convert one model to another by some rules and each wrapper don't need to take care of matching and setting all properties, this job is automatically executed by AutoMapper (AM) framework.
Final stage is a building back-end using model-view-controller (MVC) pattern on platform ASP .NET Core and as a "antagonist" on front-end side will be using "vanilla" javascript (JS). Vanilla JS means a pure language for more flexible and supportive structure. But for friendly user interface (UI) was determined a popular framework UIKit [10]. This is: "A lightweight and modular front-end framework for developing fast and powerful web interfaces" – what mentioned on their main page.
For development purposes they have also licenses and other documentation, which can be got here: https://getuikit.com/docs/.
Before start overview should be described controllers and what they do in the core application. Below is presented a part of controller diagram (fig. 10):

Figure 10. Controller's inheritance

Full version is here:
https://raw.githubusercontent.com/bladehero/LikeBusLogistic/master/LikeBusLogistic.Web/ClassDiagram.png

BaseController here is a wrapper for providing some base behavior and it's required to be inherited from [11]. So, code in this class provide a solution for getting any service the developer wants using a facade ServiceFactory, also every new action the core is trying to get identifier of user what give a basic information about anonymous access and deny it in authentication-required places.

```
public abstract class BaseController : Controller
{
    protected ServiceFactory ServiceFactory { get; set; }
    public BaseController(ServiceFactory serviceFactory) =>
ServiceFactory = serviceFactory;
```

```
    public                    override                    void
OnActionExecuting(ActionExecutingContext context)
    {
        var id = int.TryParse(context.HttpContext.User
                .FindFirstValue(ClaimTypes.NameIdentifier)
                              , out int accountId)
                 ? (int?)accountId : null;
        ServiceFactory.AccountId = id;
        ViewBag.RoleName                                    =
ServiceFactory.AccountManagement.RoleName;
        base.OnActionExecuting(context);
    }
    protected override void Dispose(bool disposing)
    {
        ServiceFactory.Dispose();
        base.Dispose(disposing);
    }
}
```

Also, one of methods is Dispose from IDisposable interface – this feature release all resources.

3. Realization and overview

This unit will be telling about how it looks like and why it was done in the way it's done. The application is crossplatform so it can be executed on different servers. Also, need to note that further will be shown the acting of microservices and their interacting with user experience and knowledge [12].
The first view the user can get is the login page. Every user can be added only through a DB and for this purpose system has got a script. So, the view is on figure 11:

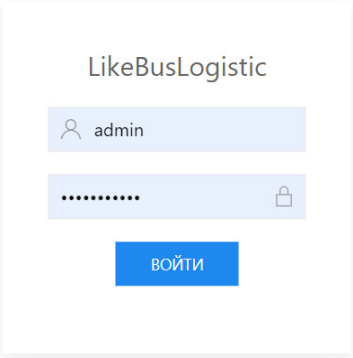

Figure 11. Login view

This page is handled by AS and all data starting from request till response stage this service supports. After successful login the user will get on the main page (fig. 12).

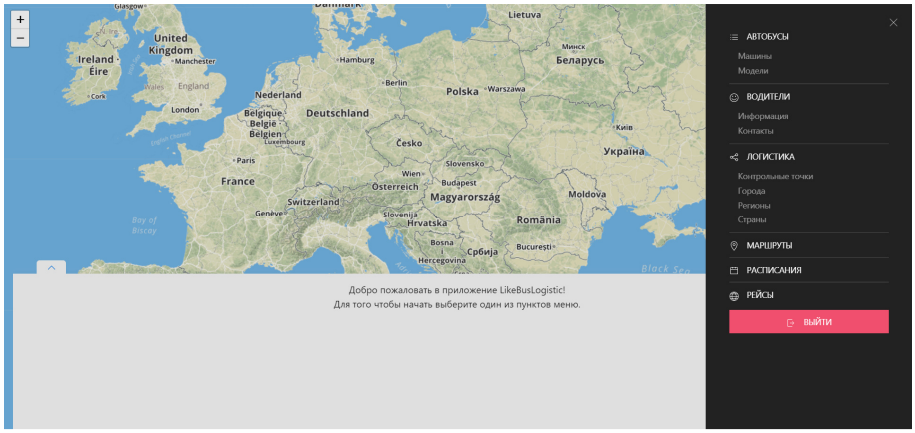

Figure 12. Main page of application

On the right side you can get a menu of points where application store and reading information:

- Buses;
- Drivers;
- Logistic;
- Routes;
- Schedules;
- Trips.

Bus and Drivers pages are used generally for saving data and connecting employees to their machines. Every partial page will use at least Create-Read-Update-Delete (CRUD) interface, but some of actions can be locked regarding to user roles. User roles in the system are three and one anonymous type:

- Unknown – for anonymous request, if action requires authenticated user then will be redirected on error page;
- Administrator – superuser rights;
- Moderator – allows to update, but delete action is blocked for some views;
- Operator – allows only read and create records.

This list may be extended on demand, but for initial goals it was enough.
Slider can have 3 positions on PC version and dynamic position on mobile phones. All application is fully stretched and adapted to different screen sizes. Minimum screen size with support starts from 330px and can be extended to 4096px. The table of buses allows to commit CRUD actions as on figure 13.

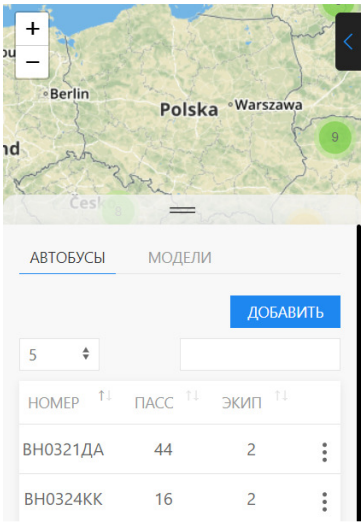

Figure 13. Bus partial view

User can change his records and affect on data using user-friendly buttons (fig. 14):

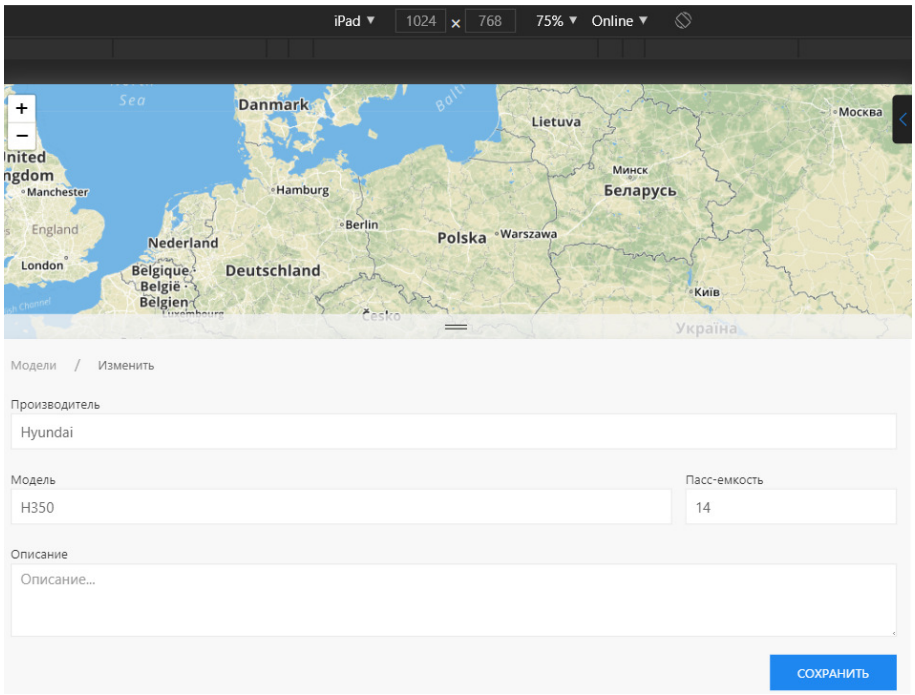

Figure 14. iPad view on changing bus models

The next menu and the most useable is “Locations” (fig. 15), here user can set a new locations for bus stops or parkings or for repairing breaks. User would need just to click on any dot on map and he will get going up slider with fields he have to enter.

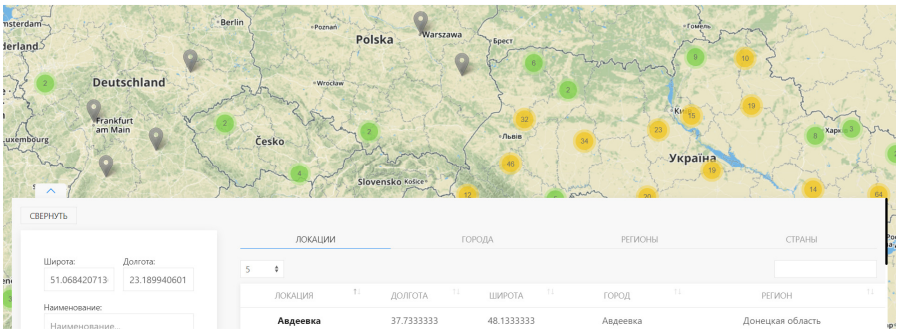

Figure 15. Locations menu

Routes menu provide a possibility to connect the locations into one array with arrival point and departures. As on figure 16 green flag is marked start and red – finish, blue points are intermediate stops.

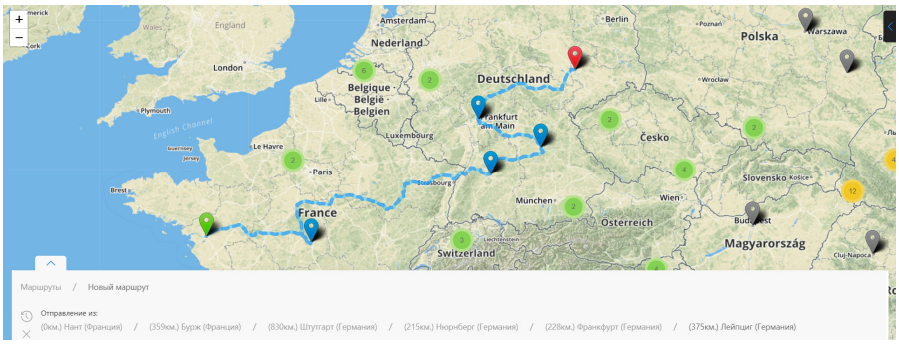

Figure 16. Routes

Auto calculating system gives full information in one time and provides estimation of distance and timings (fig. 17).

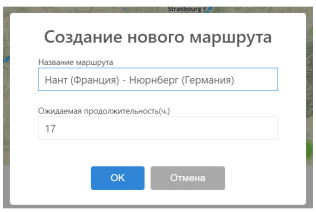

Figure 17. Creation new route

For creating time tables user should use a menu which calls Schedule and it has a design like on figure 18:

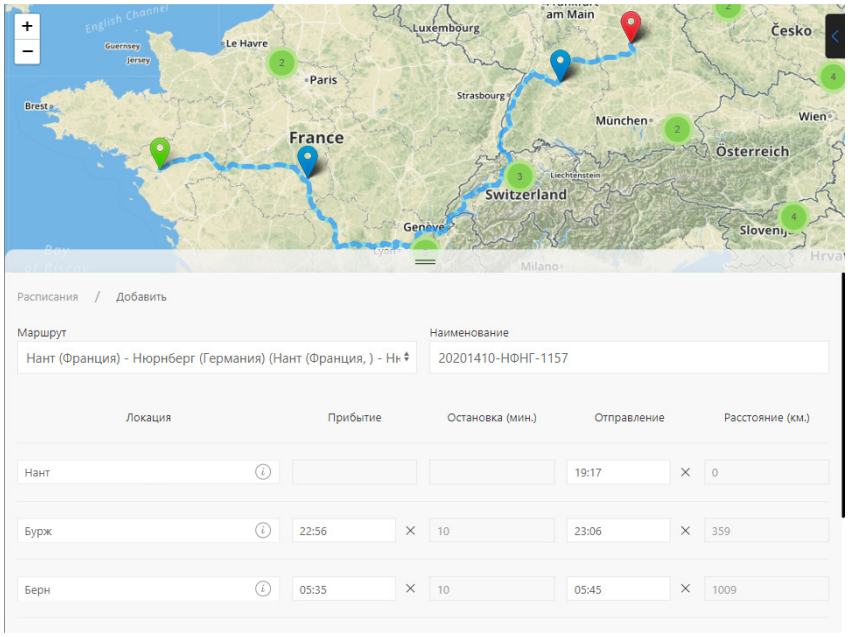

Figure 18. Time table using an existing route in the system

The final step of lifecycle is a trip management where operators have a schedules they can control using only one table and see all notifications and updates using time scheduler service (fig. 19).

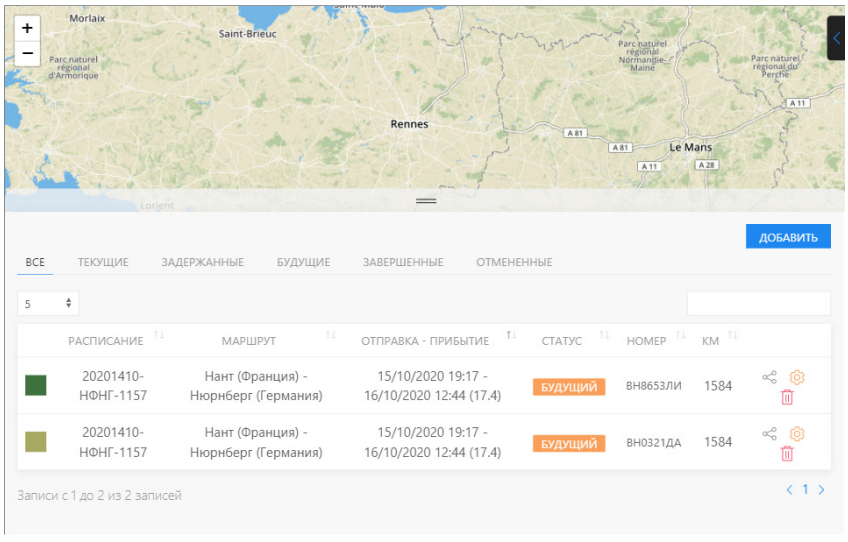

Figure 19. Trips management

4. Conclusion

In the end, several words about architecture and advantages of this application and system in general: there were a lot of actions that had to be implemented. The best decision was to split and separate them up using microservices system.

All of locations are stored in chunks and when the client move view to one of sides he will get a new tile with a new chunk with caching previous one that allows save memory and make app more reactive. Also every route user makes will be saved in data warehouse and when user will try to build a new route with some duplicated sub-routes that will cause a getting action from stored data, not making a call to external API – that saves money to customer and affect on application productivity.

The application can support in one time about 100 thousands locations and react without visible throtlings.

In the end, it should be said that every single iteration went well and totally has been finished. The project consists of all parts as was decided in the beginning. Pool of services creates a strong basement for core actions and most difficult and high-load parts were separated on standalone services. After all manual testings the system showed a progressive performance and the most possible extendable core.

This project can be reviewed by a source link:
https://github.com/bladehero/LikeBusLogistic

REFERENCES

1. SHKLAR L., ROSEN R.: Web Application Architecture. Principles, Protocols and Practices, 2009.
2. Webpage:: *https://jwt.io/*
3. Webpage: *https://developer.tomtom.com/products/directions-api*
4. C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development, 4th Edition, Mark J. Price
5. FREEMAN A.: Pro ASP.NET Core MVC 6th ed. Edition.
6. Webpage: *https://en.wikipedia.org/wiki/Design_Patterns*
7. Webpage: *https://refactoring.guru/design-patterns/facade*
8. Webpage: *https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern*
9. Webpage: *https://en.wikipedia.org/wiki/Dependency_inversion_principle*
10. Webpage: *https://getuikit.com/docs/introduction*
11. Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition, Kindle Edition

12. GODBOLT M.: Frontend Architecture for Design Systems: A Modern Blueprint for Scalable and Sustainable Websites 1st Edition.