UCN, University College of Northern Denmark
IT-Programme
AP Degree in Computer Science
dmai0919

3rd semester project - programming and technology

Alexandru Stefan Krausz, Martin Benda, Sebastian Labuda, Simeon Plamenov Kolev

21-12-2020

Title page

UCN, University College of Northern Denmark

IT-programme

AP Degree in Computer Science

Class: dmai0919

Participants:
Alexandru Stefan Krausz
Martin Benda
Sebastian Labuda
Simeon Plamenov Kolev

Supervisor: Nadeem Iftikhar

Abstract/Resume

For this semester project we have decided to take on the task of creating a system where ferry companies can advertise and sell tickets for their trips. As such we researched and didn't find a lot of good solutions available on the market. The software needed to be easily accessible to the wide public as well as fast and secure.

So we created a webapp where anyone can create an account, search and buy tickets depending on their needs as well as for companies to create a business account and post their own trips and tickets up for sale.

Normal pages/characters: 45 014 characters / 18.756 pages

Contents

Introduction 4
Problem area 4
Problem statement 4
Method and theory 4
Technology 4
Domain model 4
ORM 5
Entity Framework 6
Database-first approach 6
Code-first approach 7
Model-first approach 8
Dapper 8
ADO.NET 8
Conclusion 8
Architecture and design 9
Event-driven Architecture 9
Microservices Architecture 9
N-Tier Architecture 10
1-Tier Architecture: 10
2-Tier Architecture: 10
Our choice - 3-Tier Architecture 11
Web service 12
SOAP vs REST 12
Similarities 12
SOAP (Simple Object Access Protocol) 12
REST (Representational state transfer) 13
Our choice 14
Quick Overviews of the aforementioned communication protocols: 14

Repository pattern 15
Clients 16
Web client 16
MVC 16
MVVM 16
Dedicated client 17
WPF 17
WinForms 17
WPF vs WinForms 17
Concurrency 18
Security 20
Authentication and authorisation 20
Input validation 21
SQL Injection 21
Account Lockout 22
Testing 22
Unit Tests 22
Integration Tests 23
Groupwork 24
Conclusion 24
Literature 25
Appendices 27

Introduction

Problem area

In this semester we took a look at the ferry industry and realised there aren't too many software solutions out there for businesses to advertise and sell tickets for rides. This means there is a market gap between provider and buyer so we developed this system as a way to fill this gap and bring buyers and providers closer in a safe environment.

The system needed to be easy to use and secure, but above all else it needed flexibility in order to allow for ferry companies to sell different tickets for different trips and needs.

Problem statement

For the third semester project we decided to create a CMS software that will allow service providers to sell their products namely tickets for predetermined ferries. We feel that there aren't many great software currently on the market that are widely available for use by the general public as well as companies to reach out to as many interested "customers".

We want to harden our skills and knowledge in creating a CMS software, but this time C# will be used, in addition to a web application, which will serve as the client for the customers.

What will the developing process look like? Will we calculate developing time and be prepared correctly?
How will we solve concurrency problems, like multiple users trying to buy the same ticket?
How will we develop a web application using REST?
How will we prepare the tests, so we can ensure as least bugs as possible?
How will we address security issues, like SQL injection attacks, sensitive data exposure etc.?

Method and theory

To answer these questions, we made research about our options, different development techniques and technologies that are accessible to us.

This report will in detail answer all the questions except the first one. It will show what we have used to achieve our goal, our thoughts, considerations and reasoning behind it.

Technology

- .NET framework 4.8
- Microsoft SQL server
- Azure DevOps
- Azure Repos (git)
- Asp.NET

Domain model

Domain model is a visual representation of real-life "entities" in a model and relations between them. Each "entity" is represented as a class with attributes, which each object of a certain class possesses. It also shows which classes interact with each other.

Our domain model consists of 5 classes (Fig. 1). It all starts with the user, who beside attributes contains a role. If the role is a business account, then the user is allowed to create trips. They consist of name, further description of the trip itself, departure and arrival place and date. Trips are assigned to exactly one user. Each trip can have multiple tickets. Each ticket has its name, description, price and maximal amount. When a user is buying a ticket for a ferry, he is actually buying a UserTicket, which serves as a middle class between User and Ticket. Each User Ticket contains the first name and last name of a passenger and is assigned to exactly one User and one Ticket. This way one User can buy tickets for multiple people. UserTicket cannot exist if the Ticket does not exist and Ticket cannot exist if the Trip doesn't exist. This means User cannot buy a UserTicket if Trip does not contain any tickets. The reason why the attribute MaxTickets is included in Ticket but not in Trip is because each ferry has different capacity for cars and different for people. For this reason MaxTickets depends on the Ticket.

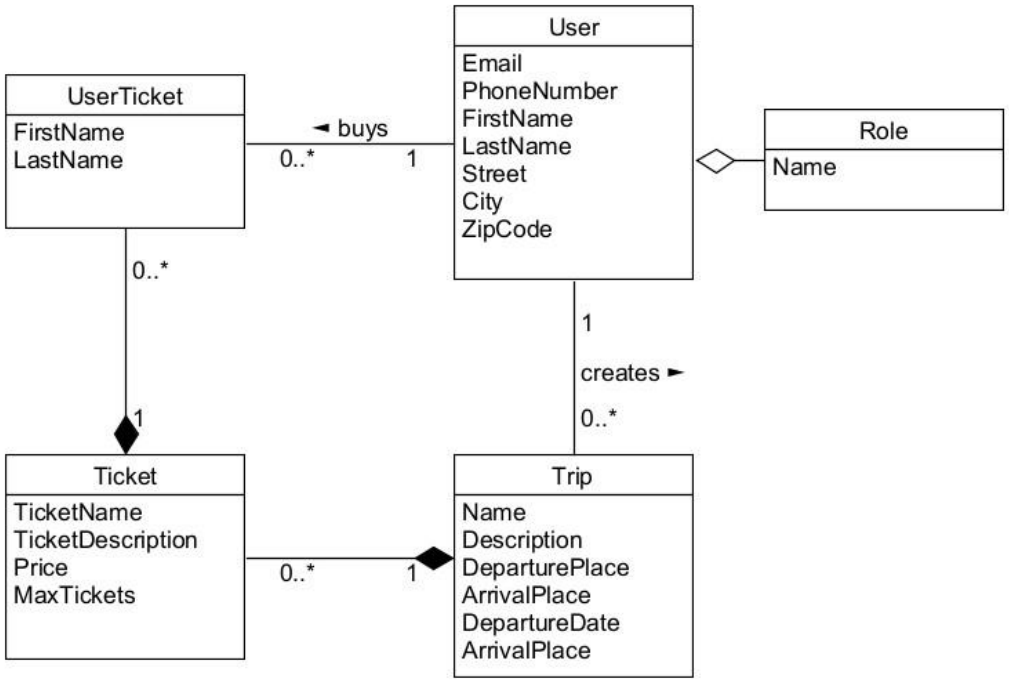

Fig. 1, Domain model

ORM

Object-relational mapping or in short ORM is a programming technique for converting data between incompatible type systems using OOP(Object-oriented programming) programming languages. [1]

Such is the case with our program, which uses C# as it's base and MySQL as the database, where each has different data types, resulting in inability to easily move data back and forth. So in order to deal with this problem, we were presented with several options for ORMs.

The way a ORM works is by creating a "virtual object database", which can then be used directly from within C#. This makes storing and retrieving data much easier and faster, resulting in overall boost in the efficiency of developing our program. [1]

Entity Framework

The ORM we decided to use is Entity Framework, which is an open-source ORM framework for ADO.NET, developed by Microsoft. Originally it was part of .NET Framework but starting with version 6 of EF, now it's delivered separately. The purpose of EF is to enable developers to work with data in the form of domain-specific objects and properties without having to concern themselves with any underlying database tables and columns, where data is being stored. [2]

With EF there are three approaches we can take:

Database-first approach

The database-first approach is the way to go, if we already have an existing Database or that we don't mind building one beforehand. After we have one of the aforementioned prerequisites, Entity Framework will then generate the Entity Designer Model XML (EDMX) and the model objects for us by using the Entity Framework Designer tool integrated into Visual Studio.

The positives about this approach is that the risk of losing data is kept at a minimum level, since any changes or updates will be performed on the Database. Also in a scenario where we already have an existing Database, this will spare us time and the need of recreating one.

Of course nothing can be perfect and so there are some negatives in going with this approach. First of all we will have limited control over the autogenerated Model classes and their source code, so we would need to have some knowledge over the EF conventions and practices in order to get exactly what we want. Secondly if we are in a situation where we have multiple instances or environments, we would have to manually update the Database of each one, which can be quite troublesome at times. Such is the case with our program, where each one of us has to manually update whenever we decide on something new, since we are using local databases.

Our choice

After some contemplation, we decided to go with this approach, but we were really torn apart between this and the code-first approach. Our main reason was that we wanted to try something new, since last year we programmed in a code-first style way. Also we wanted to have more control of the Database and set it exactly how we want and expect it. We understood the risks and problems we may encounter with the autogenerated Model classes and came to terms with it.

The first thing we did is as the name of the approach suggests - design a database. As a start we went for a simple, not 100% complete one, since we decided that we will be updating it regularly throughout development. Then using the Entity Framework Designer tool, we generated an EDMX and Model classes, which we did by providing the tool our SQL connection information. We named our EDMX - "Entities", which in turn created a new class sharing the same name, which represents the "virtual object database" mentioned a while ago. The class contains a few collections of type DbSet corresponding to the tables in the actual Database (see Fig. 2). EF allows us to directly access and manipulate the collections as if we are working with a List. So following those conventions and

practices we created the DataAccess layer, which will serve as our communication to the Database. In order to access the collections inside the “virtual database” we need to create an object of type Entities and then using that we can simply use C#’s Linq e.g. to access and take whatever we need from the database. This not only makes our code much simpler, shorter and cleaner, but also provides a great boost in the speed at which we develop the program.

```
public partial class Entities : DbContext
{
    public Entities()
        : base("name=Entities")
    {
        this.Configuration.LazyLoadingEnabled = false;
        this.Configuration.ProxyCreationEnabled = false;
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public virtual DbSet<AspNetRole> AspNetRoles { get; set; }
    public virtual DbSet<AspNetUserClaim> AspNetUserClaims { get; set; }
    public virtual DbSet<AspNetUserLogin> AspNetUserLogins { get; set; }
    public virtual DbSet<AspNetUser> AspNetUsers { get; set; }
    public virtual DbSet<Ticket> Tickets { get; set; }
    public virtual DbSet<Trip> Trips { get; set; }
    public virtual DbSet<UserTicket> UserTickets { get; set; }
```

Fig. 2, Code snippet from class Entities

Code-first approach

The Code-first approach is the flagship approach since Entity Framework 4, enabling for a highly-efficient and elegant Data Model development workflow.[3] The appeal of this is the ability for developers to define model objects using only standard classes, without needing any design tools or piles of auto generated code. In this approach, the developer first writes the Data Model entity classes and then lets Entity Framework take care of the rest - create Database only, without an EDMX.

Code-first approach is overall great for small-to-medium sized projects, as it will save a lot of time. It also allows the developer to follow a Convention over Configuration approach, to handle most common scenarios, but also gives him the chance to switch to custom, attribute-based implementation overrides, whenever he needs to modify the Database mapping.[3]

On the other hand, this approach requires a good knowledge of the ORM programming language and conventions. Furthermore, updates to the Database can be quite tricky as to not lose any data from it when migrations are being done. All of this results in a hard to maintain Database.

Model-first approach

Finally, the Model-first approach, as the name states, is where the developer creates a Model, but not just a standard Data Model class, in this case he builds a diagram. The diagram is built using the tool - Entity Designer Model XML visual interface (EDMX), which requires some experience and knowledge in working with it beforehand. After it is successfully built, EF takes care of the rest - autogenerating the Database SQL script and the Data Model source code files.

The good feature about this is the ability to create the Database schema and the class diagram as a whole using just a visual design tool, which will be very helpful especially when dealing with big and complex data structures. Additionally, whenever the database changes, the model can be updated accordingly without suffering any data loss.[3]

On the other hand, this approach is not very suitable if we want to have control over the Database and Model classes. The autogenerated SQL scripts can lead to data loss in case of updates on the diagram. Same goes for Model classes, where we have no precise control over the source code, resulting in unpredictable results at times.

Dapper

Dapper was our second option that we considered, since it offers amazing speed and performance while still having many tools for easy database manipulation. Difference from EF is that we would need to write the SQL queries ourselves, but the data conversion into an object is done automatically by Dapper. This on the other hand gives us more control over what exactly and how we are requesting from the database, which can be seen as both a pro and a con, depending on what we aim for - performance or ease of use.

ADO.NET

ADO.NET is the data access technology itself, that is used by the two aforementioned. Since it is not actually an ORM, that means everything has to be done by us. We would have to write full SQL queries, read through the data result we got back from the database by using a reader and then manually create our desired objects. The pro of this is that we have absolute control over what we send, what we receive and how we process the returned data. On the other hand, this will require much more work from our part, which may result in more errors encountered as well as insufficient time for developing the program.

Conclusion

Entity Framework takes care entirely of SQL queries and object mapping for us.
Dapper takes care of the mapping, but we have to write the SQL ourselves.
ADO.NET is the base of the other two, which means we need to take care of the SQL and object mapping ourselves.

In terms of speed EF is the slowest among the three, Dapper takes second place and ADO.NET - first, since it's working at base class level.

But nevertheless, the reason why we chose Entity Framework is because we wanted to focus more on functionality of our program without concerning ourselves that much about database manipulation. Our program in particular wouldn't suffer that much from the slower performance of Entity Framework, but in future projects where speed and performance would be of key, then using Dapper would be preferable. It also makes code much cleaner and easier to understand. Another reason why we decided to use entity framework is for learning purposes. Every company we visited was also using entity framework, so we wanted to try it and learn it.

Architecture and design

Architectures we have thought about

Event-driven Architecture

The idea behind the Event-driven architecture is that there is a central unit built that accepts all data and then delegates it to the separate modules that handle the particular type. This redirection of data is said to generate an "event", which is delegated to the code assigned to that type. A typical example of this behaviour is the browser that everyone uses and the web pages filled with Javascript that send events whenever any button is clicked. The browser itself orchestrates all of the input and makes sure that only the right code sees the right events. This kind of system is completely decoupled, since modules interact only with the events that concern them.[4]

The pros of this architecture is the adaptability to complex and often chaotic environments, it's easily scalable and extendable when new event types appear. It's a very good pattern for situations where the system works directly with the user, awaiting inputs.

As for the cons and the reason why we didn't choose this architecture is because:

1) Testing is very complex when modules affect each other. They can be tested individually, but in order to test the interaction, it has to be done in a fully functioning system.
2) Error handling is difficult to structure when several modules must handle the same events.[4]
3) Everything depends on the central unit, so if the data comes in bursts, the whole system will experience slowness. Furthermore, if a module fails, the central unit is responsible to have a back-up plan.

Microservices Architecture

This architecture is similar to the Event-driven in a way that it has it's tasks separated into different modules, but in this case - in microservices(small programs). In order for this architecture to work, the system has to contain tasks which can be easily separated and each time a new feature is implemented, it will be added as a new little program.

This architecture is suitable for big projects with many development teams and through these microservices there will be a good separation of concerns. Furthermore, it allows for rapid development of new businesses and web applications.

The cons for this architecture are the following:

1) The microservices must be largely independent or else interaction can cause the cloud to become imbalanced. Also, not all applications have tasks that can be easily split.[4]
2) Communication costs will be significantly greater, resulting also in performance loss.
3) Having that many microservices can confuse an user, since some parts of the program will be loading faster than others.

N-Tier Architecture

A N-Tier Architecture or Multilayered Architecture is a client-server architecture in which presentation, application processing and data management functions are physically separated. By doing that it provides these applications with solutions to scalability, security, fault tolerance, reusability, and maintainability. It also helps developers to create flexible and reusable applications. The only problems that are noticable is that with every tier added, the slower the system becomes and the harder it is to be implemented and maintained later.[5]

Deliberations on the N-Tier

1-Tier Architecture:

It is the simplest one as it is basically running the application on your personal computer. All of the required components for the application are run on a single server.

This type of the N-tier architecture doesn't match what we aim for at all. Our task is to build a distributed system and furthermore, the concept of our program will suffer great performance losses if it runs only on a single machine. Furthermore, anytime an update happens, the whole program will have to be redeployed, ceasing any kind of actions of users, which may trigger some negative feedback.

2-Tier Architecture:

This one is like Client-Server architecture, where communication takes place between client and server, meaning there is no middleman between the two layers.

This is a possible option for our project, but the problem will be both performance and maintaining-wise. The source code will become a mess after a while, which will slow down our development speed and also make it harder for us to find and fix errors in the system. Furthermore, everytime a new feature is implemented, both tiers will have to be redeployed, which may annoy some users. All in all the size of our idea for the project won’t cope well with a 2-tier architecture.

Our choice - 3-Tier Architecture

Since we understood that we have to create a distributed system for the current assignment, after many discussions among our group, we finally stopped at the N-Tier Architecture. Our reasoning is because it offers the best maintainability and expandability, testing and overall error handling is much easier, all the while keeping everything decoupled and nicely organised. Currently our architecture could be considered 1-Tier during development, since we are running everything on one machine. But once it's out of development and released, then it will become a 3-Tier for real.

Our architecture consists of 3 Tiers (Presentation, Business and Data Storage), which can be split into four parts: dedicated client and web application, RESTful web service, database accordingly. Since we are using the N-tier architecture, the code is separated into different layers depending on its purpose. Our database is accessed by DatabaseAccess classes in the DataAccess layer. Then we are using a repository pattern for dependency injection and easier maintainability, in the Repository layer. All our business logic is stored in the Business layer, which is called by API controllers. API controllers are the only part of the web service which is exposed to the clients. To ensure separation of concerns and high cohesion in both clients we are using Api Helpers, which take care of all calls to the API. For the dedicated client, we are using WPF (Windows presentation foundation) windows. For the web application we are using MVC (Model-View-Controller), where controllers communicate with Api Helpers and Razor Views, and those generate browser based client. All the layers have access to their model classes(see Fig. 3).

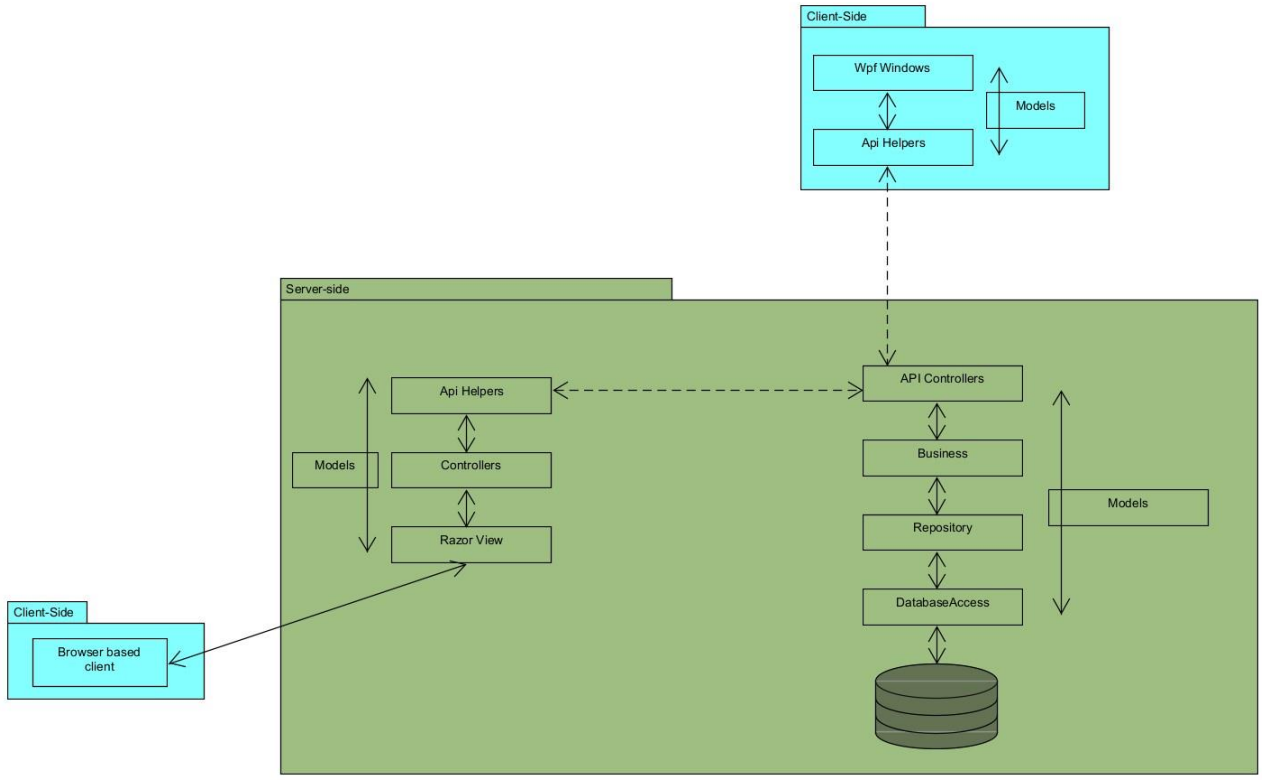

Fig. 3 Architecture of our system

Web service

SOAP vs REST

Similarities

Both SOAP and REST share similarities over the HTTP protocol, but SOAP is a more rigid set of messaging patterns than REST. In order to achieve any level of standardization in SOAP, we have to abide by the rules, whereas REST as an architecture style doesn't require processing and is naturally more flexible. Both rely on well-established rules that everyone has decided to follow in the interest of exchanging information.[6]

SOAP (Simple Object Access Protocol)

SOAP is a standardized protocol that sends messages using other protocols such as HTTP and SMTP. SOAP relies exclusively on XML to provide messaging services. After an initial release, Microsoft submits SOAP to the Internet Engineering Task Force (IETF), where it gets standardized, making it highly extensible.[6]

The three major characteristics of SOAP are:

1) Extensibility
    - SOAP was designed to support expansion, and since it's standardization, it has many acronyms and abbreviations associated with it, such as WS-Addressing, WS-Security, WS-Policy and many others. The point is that SOAP has many extensions, but you use only the ones that you need for a particular task.[6]
2) Neutrality
    - This quality allows SOAP to operate over any protocol such as HTTP, SMTP, TCP, UDP.[6]
3) Independance
    - SOAP allows for any programming language.[7]

    A very important feature of SOAP is the Built-In Error Handling, meaning if there is a problem with our request, the response will contain error information, which can be used to fix the problem. This is especially useful when we don't own the web service, otherwise we will be left guessing what's wrong. Furthermore, since the error reporting provides standardized codes, it's possible to automate some error handling tasks in our code.[6]

    Additionally, SOAP being an official protocol, comes with strict rules and advanced security features such as built-in ACID compliance and authorization.[8]

Drawbacks of SOAP:

- It has a bigger learning curve than REST. It mostly depends on the programming language, since in some, the requests have to be built manually and that can be quite problematic, because SOAP is intolerant to errors.
- Slower and less-efficient than REST, because it requires extensive processing and since it is using XML for all messages, they are bigger and their structure has to be created every time,

compared to the ones REST deals with. Furthermore it requires more bandwidth and resources, leading to slower page load times.

REST (Representational state transfer)

REST is a software architectural style that defines a set of constraints used for creating Web Services. A Web Service that uses REST is called a RESTful Web Service, providing interoperability between computer systems on the internet. It allows the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations. Since RESTful systems are using a stateless protocol and standard operations, they aim for fast performance, reliability, and the ability to expand reusing modules, which can be managed without even affecting the system while running.[8]

In order to create a REST API, a set of six architectural constraints have to be followed. By operating within these constraints, the system gains many non-functional properties, such as performance, scalability, simplicity, etc.

1) Uniform interface
    - Requests from different clients should look the same, e.g. the same resource shouldn't have more than one URI.[9]
2) Client-server separation
    - Client and server should act independently and interaction should happen only through requests and responses.
3) Statelessness
    - There shouldn’t be any server-side sessions and each request should contain everything the server needs to know.[9]
4) Cacheable resources
    - Server responses should state if the data they send is cacheable or not. Cacheable resources should arrive with a version number, which can be used in order to avoid requesting the same data once again.[9]
5) Layered system
    - There might be several layers of servers (proxies) in between the client and server, but that shouldn’t affect either the request or response.[9]
6) Code on demand
    - When it’s necessary, the request can contain executable code, e.g. Javascript within an HTML response.[9]

Advantages:

- Learning curve is very small, because instead of creating whole XML structures like in SOAP, REST relies on a simple URL and can use the four different HTTP 1.1 verbs to perform tasks.
- The output you get out of requests is in a form that is easy to parse within the language you’re using for your application, e.g JSON, CSV.
- It’s fast since it doesn’t require any extensive processing.
- It’s efficient, because REST uses smaller message formats, compared to SOAP.

Disadvantages:

- It's not as language, platform and transport independent as SOAP, since REST requires the use of HTTP.
- Doesn't have built-in error handling, ACID compliance and authorization.

Our choice

Since this is our first time developing a distributed system with a Web Service and separate clients, we decided to go with REST, which has a smaller learning curve compared to SOAP. Also the message formats, like JSON, are easier to work with. All the while, they are smaller and more efficient than SOAP's XML.

Quick Overviews of the aforementioned communication protocols:

Application Layer

HTTP (Hypertext Transfer Protocol)

HTTP is a protocol that allows the fetching of resources, such as HTML documents. It is a client-server protocol, which means requests are initiated by the recipient, the Web browser e.g. Clients and servers communicate by exchanging individual messages (not a stream of data). The messages sent from the client (user-agent) are called requests and the ones from the server are responses. In between the two there are numerous entities, collectively called proxies, which perform different operations and act as gateways or caches. (e.g routers, modems, etc.) (see Fig. 4). [10]

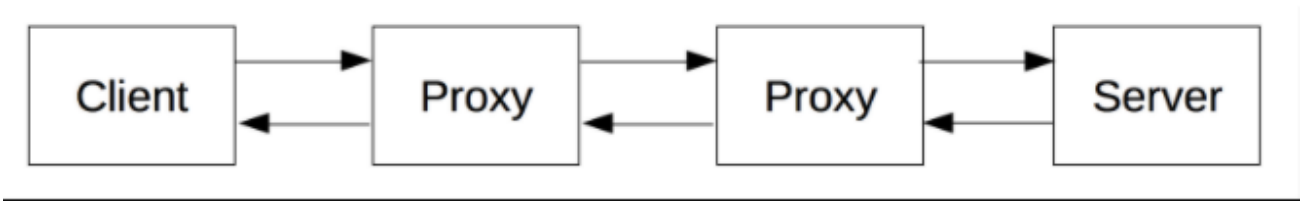

Fig. 4, Proxies [10]

Furthermore, HTTP is an application layer (abstraction layer that specifies the shared communication protocols and interface methods used by hosts in a communications network) protocol, that is sent over TCP and stands at the top. (see Fig. 5). [10]

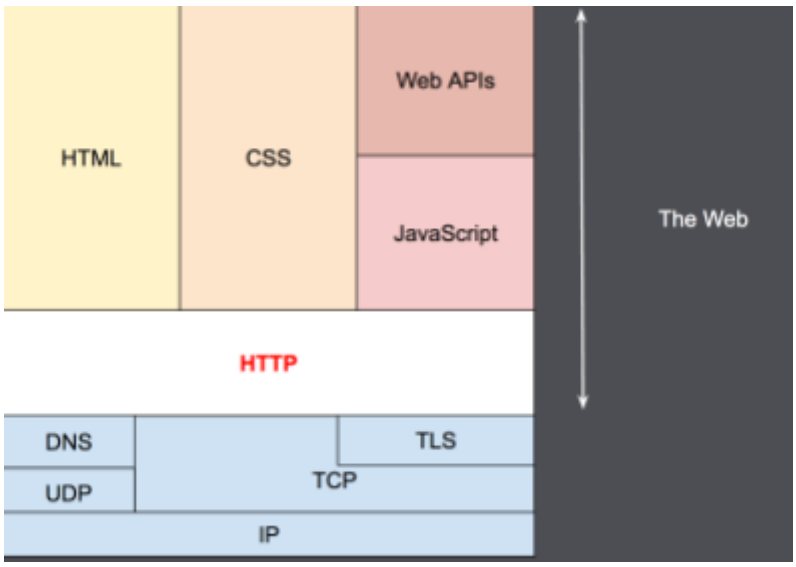

Fig.5 [10]

SMTP (Simple Mail Transfer Protocol)

SMTP is a connection-oriented (first a connection has to be established, before data can be sent), text-based protocol (in a human readable format) in which a mail sender communicates with a mail receiver by sending command strings and supplying necessary data over a reliable data stream, like TCP.[11]

Transport Layer

TCP (Transmission Protocol)

TCP is one of the main protocols of the Internet Protocol Suite. It complements the IP (Internet Protocol), that's why it is referred to as TCP/IP. It provides reliable, ordered, and error-checked delivery of a stream of bytes between applications running on hosts that communicate via an IP network. It is connection-oriented, and a connection between client and server is established before data can be sent.[12]

UDP (User Datagram Protocol)

UDP is one of the core members of the Internet Protocol Suite. With UDP, computer applications can send messages to other applications, in this case referred to as datagrams. No prior connections need to be established, since UDP uses a simple connectionless communication model with a minimum of protocol mechanisms. Compared to TCP, UDP is unreliable, since it doesn't perform error-checkings and corrections. If the priority is time over reliability, then UDP should be chosen.[13]

Repository pattern

Repository pattern provides an abstraction of data, so that our application can work with a simple abstraction that has an interface similar to that of a collection. It contains a series of straightforward methods that don't deal with database concerns like connections, commands, etc. Through this pattern we can achieve loose coupling and keep domain objects persistence ignorant.[14]

For our system we went for the one-repository per entity approach, each one of them containing simple methods that connect the controllers with the data access layer. Our main reason was because of decoupling and separation of concerns.

Clients

Web client

MVC(Model View Controller) pattern in which Views access controllers and controllers access Models, this offers separation of concerns. MVVM(Model View ViewModel) pattern offers two-way data binding between view and view-model. The view-model makes use of the observer pattern to make changes in the view-model.

MVC

Pros: -
Development of the application becomes fast.
-Easy for multiple developers to collaborate and work together.
-Easier to Debug as we have multiple levels properly written in the application.
Cons: -
It is hard to understand the MVC architecture. -
Must have strict rules on methods.[15]

MVVM

Pros :
-MVVM facilitates easier parallel development of a UI and the building blocks that power it.
-MVVM abstracts the View and thus reduces the quantity of business logic (or glue) required in the code behind it.
-The ViewModel can be easier to unit test than in the case of event-driven code.

Cons:
-For simpler UIs, MVVM can be overkill.
-While data bindings can be declarative and nice to work with, they can be harder to debug than imperative code where we simply set breakpoints.
-In larger applications, it can be more difficult to design the ViewModel up front to get the necessary amount of generalization.[16]

Based on that we decided that MVC pattern would be better suited for our needs during this project.

Using our system a user is able to buy tickets for ferry rides. The tickets can differ depending on your needs. Also business users are able to create their own trips and range of tickets as well as their quantity. Business accounts will be reserved for companies that will only be able to edit their respective trips and tickets. Only admin accounts have access to trips that do not belong to them.

Dedicated client

WPF

WPF, which stands for Windows Presentation Foundation, is known as Microsoft's latest approach to GUI framework, used alongside the .NET framework. The core of WPF is a resolution-independent and vector-based rendering engine that is built to take advantage of modern graphics hardware. WPF's main strength is in offering valuable application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, 2D and 3D graphics, animation, styles and much more. As mentioned already before, WPF is part of .NET, creating the possibility of building applications that incorporate other elements of the .NET API. [17]

With WPF, we are capable of developing an application using both mark as well as code-behind, general function known by developers with some previous experience. Just as expected, HTML is used to implement the visual side/appearance while using programming languages to implement its behaviour. Thank to that, wpf has a lot of benefits to offer, mainly:

- Overall development and maintenance costs are reduced due to low coupling

- Development is more efficient because designers can implement an application's appearance simultaneously with developers who are implementing the application's behavior

WinForms

Window Forms, just as WPF is a UI framework for building Windows desktop apps. This framework was generally well known for providing one of the most productive ways to create desktop apps based on the visual designer provided in Visual Studio. Thanks to its functionality (for example drag-and-drop placement) of visual controls makes it easy to build desktop apps. What is more, WinForms has a very rich usage of toolbars, menus, display of submenus, text boxes and combo boxes. [18]

WPF vs WinForms

Both WPF and WinForms offer very specific advantages over each other. Both of these platforms offer all kinds of functions that could be suitable for our project. After multiple discussions, we have decided to use WPF for our project. Main reason why we decided to go with WPF was mostly due to specializations and usage.

In this part, we will take closer look at advantages of these platforms:

WPF Advantages:

- It's newer and much more flexible

- XAML makes it easy to create and edit your GUI as well as allowing the work to be split between a designer and a programmer
- Databinding, which allows you to get a more clean separation of data and layout [19]

WinForms Advantages:

- It's older and therefore tested by time, much more reliable
- There's greater support in the 3rd-party control space for WinForms
- The designer in Visual Studio is still, as of writing, better for WinForms than for WPF, where you will have to do more of the work yourself with WPF (WPF not supported cross-platform usage) [20]

Since we already started to talk about dedicated client as well as wpf, it is in place to mention the basic functions of wpf that has been implemented for our system. WPF is mainly aimed for the purposes of the admin and all privileges that come with this role. As may be expected, admin won't be allowed to buy tickets through the WPF. On the other hand, he will have power to create, delete or correct incorrect information for any user. It is mostly due to the fact that there can be situations in which a defect was done and the admin will have privileges to correct these mistakes.

Concurrency

Concurrency means multiple computations are happening at the same time.[21] It especially occurs when multiple users are trying to access the same data, which mostly happens in applications like ours, where many users can access and change data. In our case, concurrency problems arise when multiple users are buying the same ticket. Unhandled, users would be able to buy more tickets than available. There are two common approaches for solving concurrency issues: optimistic approach and pessimistic approach.

In the optimistic approach we assume that the probability of concurrency problems is very low. While updating data in the database, the application must check if the data was modified since it was read. This can be accomplished by having a row version field in the database or date, when it was last modified. Advantages of an optimistic approach is especially responsiveness and performance. Entity framework has built-in optimistic concurrency and will take care of it, when the row version is added and set.

Pessimistic concurrency on the other hand expects often conflicts and uses locks. By locking rows in databases or methods in code, no other user can modify/access a certain part until the lock is released. To increase the effectiveness there are different locks depending on what the user is doing with the data. For example, if there are multiple users, who are only reading the data, there is no reason to restrict them access since the data will not change. This locking is secured by different isolation levels. Although they prevent conflicts from happening, their drawback is lower system responsiveness. The stronger the isolation level, the lower is the responsiveness. They need to be chosen carefully in order to stabilize the system. Using locks creates another problem, namely a chance of getting into deadlock.

Considering how our system is built, we decided to use a pessimistic approach. In our system, when a user is buying tickets, we are not updating any rows in the database, we are only adding rows to the table UserTicket. As can be seen in figure 6, we are wrapping the whole buying into the transaction. It means that if an error occurs at any time, the buying won't be successful and the transaction will be rolled back. This way we can ensure consistent data in the database and that the user will buy either all the tickets they selected or none. This is also a place where we use our lock. We use generic static objects (see Fig. 7) and C# lock statement. The lock statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released.[22] First we count the tickets that were already bought and compare them to the allowed maximum amount. For that we created an entry class which consists of ticketId and amount, which the user wants to buy. If the amount the user wants to buy is bigger than the current amount they can buy, DbUpdateConcurrencyException is thrown, transaction rollbacked and proper message sent to the user (HttpStatusCode 409 Conflict). If the comparison is okay, then all first names and last names are added to the UserTicket table. If a problem occurs while adding rows to the UserTicket table, an exception is thrown, transaction rollbacked and again a proper message sent to the user (HttpStatusCode 500 Internal Server Error). If everything went right, changes are saved and transactions committed.

Fig. 6, Code snippet

```
1 reference | Sebastián Labuda, 13 hours ago | 2 authors, 8 changes
public bool BuyTickets(List<UserTicket> userTickets, List<TicketAmountEntry> entries)
{
    using (var dbContextTransaction = dbContext.Database.BeginTransaction())
    {
        lock (BuyingLock)
        {
            try
            {
                foreach (var entry in entries)
                {
                    var amountBought = dbContext.UserTickets.Count(t => t.TicketId == entry.TicketId && t.Active == true);
                    var amountMax = dbContext.Tickets.First(t => t.Id == entry.TicketId).MaxTickets;

                    if (amountMax - amountBought < entry.Amount)
                    {
                        throw new DbUpdateConcurrencyException();
                    }
                }

                foreach (var userTicket in userTickets)
                {
                    try
                    {
                        dbContext.UserTickets.Add(userTicket);
                    }
                    catch (Exception ex)
                    {
                        throw new Exception();
                    }
                }
                dbContext.SaveChanges();
                dbContextTransaction.Commit();

                return true;

            }
            catch (DbUpdateConcurrencyException ex)
            {
                dbContextTransaction.Rollback();
                throw ex;
            }
            catch (Exception)
            {
                dbContextTransaction.Rollback();
                return false;
            }
```

```
private static readonly object BuyingLock = new object();
```

Fig. 7

Using a static object as a lock means that no matter which ticket the user is buying, they will be waiting if another user is currently inside the transaction. This raised our concerns if this solution is usable in real life and if the performance won't suffer too much. To ensure its scalability, we used Apache JMeter for load testing to see how our program behaves if many people will try to buy at the same time. But even in extreme scenarios when 100 users are buying at the exact same time, the average time of buying was 0,574 seconds. (see Fig. 8 and 9). Even in the worst case it took only 0,765 seconds, which is more than acceptable.

Label	# Samples	Average	Min	Max	Std. Dev.
HTTP Reque...	100	574	370	765	81.56
TOTAL	100	574	370	765	81.56

Fig. 8, Load test result, summed

Sample #	Start Time	Thread Name	Label	Sample Time(...	Status	Bytes	Sent Bytes	Latency	Connect Tim...
1	12:34:45.458	Thread Grou...	HTTP Request	370		396	988	370	158
2	12:34:45.447	Thread Grou...	HTTP Request	439		396	988	439	167
3	12:34:45.464	Thread Grou...	HTTP Request	428		396	988	428	129
4	12:34:45.459	Thread Grou...	HTTP Request	438		396	988	438	157
5	12:34:45.461	Thread Grou...	HTTP Request	443		396	988	443	179
6	12:34:45.450	Thread Grou...	HTTP Request	465		396	988	465	166
7	12:34:45.447	Thread Grou...	HTTP Request	481		396	988	481	170
8	12:34:45.470	Thread Grou...	HTTP Request	466		396	988	466	150
9	12:34:45.460	Thread Grou...	HTTP Request	482		396	988	482	115
10	12:34:45.469	Thread Grou...	HTTP Request	479		396	988	479	106
11	12:34:45.470	Thread Grou...	HTTP Request	483		396	988	483	166
12	12:34:45.447	Thread Grou...	HTTP Request	513		396	988	513	169
13	12:34:45.447	Thread Grou...	HTTP Request	520		396	988	520	170
14	12:34:45.449	Thread Grou...	HTTP Request	525		396	988	525	178
15	12:34:45.472	Thread Grou...	HTTP Request	520		396	988	520	156

Fig. 9, Load test result, samples

Security

Authentication and authorisation

Authentication and authorisation are referring to two different processes used in order to secure an application. Authentication refers to the process of validating your credentials such as username/userid and password in order to verify your identity whereas authorisation comes after authentication and refers to the system deciding whether or not you have access to a certain resource.

When it comes to authentication we are using a username and password combination in order for our users to login in our system. When it comes to security we are using hashed passwords to prevent easy access for hackers and account lockout to prevent brute force attempts and security stamps to log out users from the application when the password or associated login credentials have been modified. For authorisation we are using authorised methods and user roles to limit a specific users access to the system.

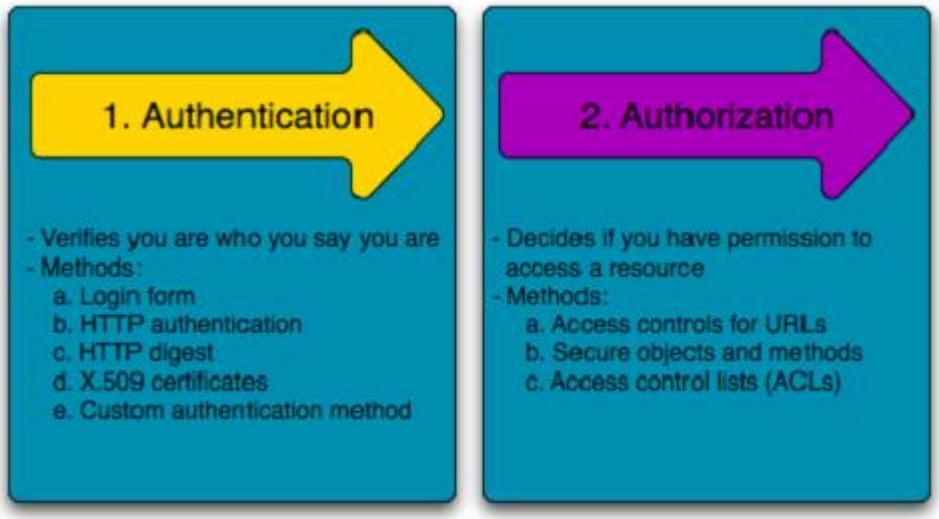

Fig 10, [23]

Input validation

User inputting wrong data is a problem of every system. It can lead to invalid data in the database or worse to crash of the whole system. That's why it's important to always check what user inputs. Checks can be done in the frontend and backend.

In the web client in the front end it is done by using javascript(for example jQuery). Even though it is good to validate input this way, because it will save some processing time in the backend, since it is in the user's browser, knowledgeable people can rewrite them, so relaying on them is not a good idea.

Once the input reaches the API, we are validating the model of the object. In case of wrong input, the API returns http status code 400 “Bad Request”. For that we are using data annotations in the model classes. But built-in annotations don't have enough options, that's why we are using NuGet package FoolProof, especially in the class Trip(see Fig. 11).Besides that the fields are required, we are also comparing departure and arrival date, so the arrival date always has to be bigger than departure date.

```
[Required]
[DataType(DataType.DateTime)]
7 references | Simeon Plamenov Kolev, 7 days ago | 2 authors, 2 changes
public System.DateTime DepartureDate { get; set; }
[Required]
[DataType(DataType.DateTime)]
[GreaterThan("DepartureDate")]
5 references | Simeon Plamenov Kolev, 7 days ago | 2 authors, 2 changes
public System.DateTime ArrivalDate { get; set; }
```

Fig. 11, Code snippet

```
public IHttpActionResult PostTrip(Trip trip)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
```

Fig. 12, Code snippet

SQL Injection

SQL injection is a vulnerability that allows attackers to manipulate the database of the system. If the prevention against SQL injection is not in place, users can write additional input to execute different code in the database. In such cases they can access accounts of different users, change data or completely delete tables and the whole database. In our case, the entity framework is

using LINQ to Entities. The queries are not created by using string and combining strings, but through an object model. This way, the application is protected from SQL injections and attacks are no longer possible.

Account Lockout

One of the methods hackers are using to get another user's credentials is brute force. Brute force attack means submitting as many password combinations as possible, in hope that the correct password will be found. Since many users are using simple or known passwords, this method is widely used. In order to prevent brute forcing, we are using account lockout. Every time a user tries to login and inserts the wrong password, the counter AccessFailedCount in the database tied to the account will increment by one. After 6 wrong attempts, the account will be locked out and the user will not be able to login for the next 5 minutes. After locking the user, the counter resets back to 0. If the user fails multiple times to login and manages to login after that, the counter will reset to 0 as well. Since brute force is trying as many passwords as possible, this way we can at least delay this attempt for a long amount of time.

LockoutEndDateUtc	LockoutEnabled	AccessFailedCount
NULL	1	0
NULL	1	2
2020-12-17 11:58:46.097	1	0
NULL	1	0

Fig. 13

Testing

In order to ensure the delivery of a high quality product we decided to use the test driven development approach for our backend and live tests for the front end.

Unit Tests

Unit tests are tests made for individual units of code or methods. These tests are usually automated and have the goal of ensuring that a section of the application meets its design and behaves as intended.

Because of our current project's lack of complex individual methods we decided to Unit test only the Count method used for buying tickets.

```
[TestMethod]
0 references | 0 changes | 0 authors, 0 changes
public void CountUnitTest()
{
    UserTicketManagement target = new UserTicketManagement();
    PrivateObject obj = new PrivateObject(target);
    UserTicket userTicket1 = new UserTicket("Alex", "K. Stefan", 1, "9643f2db-5743-494d-891b-5d4faa8c4545", true);
    UserTicket userTicket2 = new UserTicket("Alex", "K. Nicola", 2, "9643f2db-5743-494d-891b-5d4faa8c4545", true);
    UserTicket userTicket3 = new UserTicket("Alex", "S George", 3, "9643f2db-5743-494d-891b-5d4faa8c4545", true);
    UserTicket userTicket4 = new UserTicket("Alex", "S George", 1, "9643f2db-5743-494d-891b-5d4faa8c4545", true);
    UserTicket userTicket5 = new UserTicket("Alex", "S George", 2, "9643f2db-5743-494d-891b-5d4faa8c4545", true);
    List<UserTicket> UserTickets = new List<UserTicket>();
    UserTickets.Add(userTicket1);
    UserTickets.Add(userTicket2);
    UserTickets.Add(userTicket3);
    UserTickets.Add(userTicket4);
    UserTickets.Add(userTicket5);
    var retVal = (List<TicketAmountEntry>)obj.Invoke("Count", UserTickets);
    bool result;
    if(retVal.Count == 3)
    {
        result = true;
    }
    else
    {
        result = false;
    }
    Assert.IsTrue(result);
}
```

Fig. 14, Code snippet

Integration Tests

Integration testing are tests where multiple methods and units of code are tested in conjunction with each other in order to evaluate the system in conformity with specific functional requirements.

Due to the nature of this semester's project our group came to the conclusion that unit testing will not bring us much benefit in comparison with the time required for it so in order to ensure a high quality system we decided to do Integration tests.

```
[TestMethod]
0 references | Alexandru Stefan Krausz, 1 hour ago | 1 author, 1 change
public void GetAllUserTicketsTest()
{
    IEnumerable<UserTicket> result = userTicketManagement.GetAllUserTickets();
    Assert.IsNotNull(result);
}

[TestMethod]
0 references | Alexandru Stefan Krausz, 1 hour ago | 1 author, 1 change
public void GetUserTicketByIdTest()
{
    UserTicket temp = userTicketManagement.GetUserTicketById(1);
    bool result;
    if(temp.FirstName == "Alex" && temp.LastName == "K. Stefan")
    {
        result = true;
    }
    else
    {
        result = false;
    }
    Assert.IsTrue(result);
}


[TestMethod]
0 references | Alexandru Stefan Krausz, 1 hour ago | 1 author, 1 change
public void buyUserTicketTest()
{
    UserTicket TestUserTicket = new UserTicket("Alex", "N. Peter", 4, "9643f2db-5743-494d-891b-5d4faa8c4545", true);
    List<UserTicket> tempList = new List<UserTicket>();
    tempList.Add(TestUserTicket);
    userTicketManagement.BuyTickets(tempList);
    tempList = userTicketManagement.GetUserTicketsByUserId("9643f2db-5743-494d-891b-5d4faa8c4545");
    bool result = false;
    foreach(var item in tempList)
    {
        if(item.FirstName == "Alex" && item.LastName == "N. Peter" && item.TicketId == 4)
        {
            result = true;
        }
    }
    Assert.IsTrue(result);
}
```

Fig. 15, Code snippet

Groupwork

Our group has developed during the past few semesters. Teamwork as well as fluent work has become truism for our group. Understanding our stronger sides alongside weaker ones helped to correctly divide the work between all group members while reassuring the attaining general knowledge from different areas of work.
Regardless of all of these facts and previous knowledge, we have been once again challenged by the current state of Corona-19 pandemic situation all around the world. As anyone else, even our group in both terms of studying, along with working on our project has been heavily influenced. Our group had to face both already well known problems as well as to face completely new ones. Regarding the project, the biggest challenge proved to be the front-end this time. We have been working in a completely new environment and it understandably took some time to get used to. On the other hand, regardless of working with different programming languages, we could see advancement in personal back-end skills of every group member. The time spent on the back-end was rapidly lowered, while we had to put more focus into the front-end part.
On the other hand, we stepped towards the whole situation with maximal respect, motivation and understanding. Regardless of group members being sick during multiple sprints, partial loss of motivation or interest, each member put as much effort into this project as possible. Each member of the group worked the same way, including any part of our project.
To sum up, pandemic served as a great test. We had a chance to use all our previous as well as newly gained knowledge and studying materials we have learned both semester and even increase our knowledge.

Conclusion

Overall we have met the requirements and have reached our goal. We implemented most of the user stories we prepared at the beginning. In this project we used and tried loads of things for the first time and even though we were working with many unknowns, we still managed to finish it and learn a lot. We managed to create a functioning system where users can buy tickets for ferry rides and companies advertise them.

Although we implemented a lot, our system is not completed. There are still some user stories and tweaks especially regarding usability that can be done. Adding functionality to dedicated client as well as security considerations into web client could improve the quality of our project.

This project helped us to dig deep into new methods and technologies. We learned a lot about different ORMs, web services and web clients, their advantages as well as disadvantages. We now better understand security related problems regarding to web services/applications and how to deal with them.

Literature

[1] en.wikipedia.org , [Online].

https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping

[Accessed: 17-Dec-2020].

[2] en.wikipedia.org , [Online].

https://en.wikipedia.org/wiki/Entity_Framework

[Accessed: 17-Dec-2020].

[3] ryadel.com , [Online].

https://www.ryadel.com/en/code-first-model-first-database-first-vs-comparison-orm-asp-net-core-entity-framework-ef-data/

[Accessed: 17-Dec-2020].

[4] techbeacon.com , [Online].

https://techbeacon.com/app-dev-testing/top-5-software-architecture-patterns-how-make-right-choice

[Accessed: 17-Dec-2020].

[5] guru99.com , [Online].

https://www.guru99.com/n-tier-architecture-system-concepts-tips.html

[Accessed: 17-Dec-2020].

[6] smartbear.com , [Online].

https://smartbear.com/blog/test-and-monitor/soap-vs-rest-whats-the-difference/

[Accessed: 18-Dec-2020].

[7] en.wikipedia.org , [Online].

https://en.wikipedia.org/wiki/SOAP

[Accessed: 18-Dec-2020].

[8] en.wikipedia.org , [Online].

https://en.wikipedia.org/wiki/Representational_state_transfer

[Accessed: 18-Dec-2020].

[9] raygun.com , [Online].

https://raygun.com/blog/soap-vs-rest-vs-json/#differences

[Accessed: 18-Dec-2020].

[10] developer.mozilla.org , [Online].

https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[Accessed: 18-Dec-2020].

[11] en.wikipedia.org , [Online].

https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol

[Accessed: 18-Dec-2020].

[12] en.wikipedia.org , [Online].

https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[Accessed: 18-Dec-2020].

[13] en.wikipedia.org , [Online].

https://en.wikipedia.org/wiki/User_Datagram_Protocol

[Accessed: 18-Dec-2020].

[14] deviq.com , [Online].

https://deviq.com/repository-pattern/

[Accessed: 20-Dec-2020].

[15] interserver.net , [Online].

https://www.interserver.net/tips/kb/mvc-advantages-disadvantages-mvc/

[Accessed: 19-Dec-2020].

[16] oreilly.com , [Online].

https://www.oreilly.com/library/view/learning-javascript-design/9781449334840/ch10s07.html

[Accessed: 19-Dec-2020].

[17] docs.microsoft.com , [Online].

https://docs.microsoft.com/en-us/visualstudio/designers/getting-started-with-wpf?view=vs-2019

[Accessed: 20-Dec-2020].

[18] docs.microsoft.com , [Online].

https://docs.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-5.0

[Accessed: 20-Dec-2020].

[19] wpf-tutorial.com , [Online].

https://www.wpf-tutorial.com/about-wpf/wpf-vs-winforms/

[Accessed: 20-Dec-2020].

[20] stackoverflow.com , [Online].

https://stackoverflow.com/questions/31154338/windows-forms-vs-wpf

[Accessed: 20-Dec-2020].

[21] web.mit.edu , [Online].

https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/

[Accessed: 16-Dec-2020].

[22] docs.microsoft.com , [Online].

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement

[Accessed: 16-Dec-2020].

[23] medium.com , [Online].

]https://medium.com/datadriveninvestor/authentication-vs-authorization-716fea914d55

[Accessed: 19-Dec-2020].

Appendices