VIA University College

Semester Project Report

Dining with strangers

Tamas Fekete (266931 ICT)
Dorcia Fiona Dinesh (280145 ICT)
Zuzanna Maria Czepukojc (280163 ICT)
Lau Ravn Nielsen (280337 ICT)

Supervisor:

Jakob Knop Rasmussen (JKNR)
Jan Munch Pedersen (JPE)
Software Engineering
3rd semester

Table of Contents

Abstract

This project was executed based on the idea of connecting people through dining events application in order to reduce the effects of loneliness. This application is built using technologies like Blazor .NET in C# for the graphical user interface as the first tier, REST API in Java spring framework as second tier, LINQ .NET in C# with database SQLite as the third tier, HTTP. The software tools used to implement the project in a heterogeneous system of three tiers are Rider IDE for the first and third tier. IntelliJ IDE was used for the second tier to implement business logic. The application passed a few tests and is not completely implemented according to its requirements. There are a lot of opportunities for improvements in the future for this project.

1. Introduction

Humans have evolved into social beings out of the necessity of being dependent with each other in order to survive harsh circumstances. In today's world as well, people continue to have a need to associate with others. Finding new ways to develop strong social connections is integral to the wellbeing of humans. The lack of such connections can lead to many issues, including loneliness. Loneliness in its essence is a negative feeling one may experience when our social needs aren't met by our current social circle (Blackdoginstitute.org.au, 2018). The causes for loneliness fluctuate over the duration of one's life. This is due to the different needs at different ages (Department for Digital, Culture, Media and Sport, 2019).

However, a few causes could be one's personality, circumstances, environmental conditions, local social networks, social media, the way we live and work, or life events (Department for Digital, Culture, Media and Sport, 2019).

Loneliness is often described as an epidemic (Hrsa.gov, 2019). It's described as an epidemic due to the large share of the population which is affected by it. In a 2018 survey from Australia 50 percent of the respondents reported feeling lonely (Dingle, 2019). In Denmark, 12% of young people in the age group 16-24 years feel very lonely on the UCLA Loneliness Scale, with the majority being women (15%) and men (10%) (Lasgaard, Christansen and Friis, 2019). A recent study of adults of all ages in San Diego, California concluded that 76% of the respondents felt moderately to high levels of loneliness on the UCLA-3 scale (Ardelt and Ferrari, 2018).

Whenever one begins to feel lonely, he or she will experience an increased feeling of vulnerability, which in turn will take a toll on both the body and mind (Blackdoginstitute.org.au, 2018). There's significant evidence supporting the link between various mental problems such as high-stress levels, anxiety, poor sleep quality, self-harm, depression, and extended periods of psychological suffering (Lasgaard, Christansen and Friis, 2019) (T. Cacioppo and Elizabeth Hughes, 2006). Loneliness increases the risk for dementia, likely through these mechanisms, however, the absence of social interaction itself may also be a primary factor in that social stimulation can help maintain brain health (Cacioppo & Hawkley, 2009; Cacioppo et al., 2014).

However, it is not clear whether or not loneliness is the cause or the effect of the aforementioned mental problems. Nevertheless, it's clear that loneliness goes hand-in-hand with various mental problems (Lasgaard, Christansen and Friis, 2019).
A few ways to combat the feeling of loneliness, is introspection, connection and interaction. Meeting and interacting with strangers, or interacting with people one already knows, is the key to fighting one's loneliness (Carlton, 2019). One of the few initiatives the Danish government has deployed in 2016 in its fight against loneliness, was Denmark-eats-together (MaryFonden.dk,

2016). Before launching the campaign, Maryfonden conducted a survey in order to explore the waters beforehand. The survey showed that 23,6% of the respondents missed someone to dine with, and 44% of the respondents would dine with someone they barely know (Korsgaard, 2016). Socializing has various impacts on the human body such as brain health, mental health and it may lead to a stronger immune system. It is proven by a 2017 study from Dublin, Ireland that after the age of 50 more socialized individuals had better processing speed, memory, verbal fluency and emotional regulation. Furthermore, it may reduce the risk of dementia and symptoms of Alzheimer's more frequently than the nonsocial counterparts.

For mental health, socializing may help to develop a sense of belonging and other supportive feelings and beliefs which is something both introverts and extroverts wants to achieve. When people feel better self-esteem, it may help one reduce the likeliness of anxiety disorders and depression. It is proven that laughter itself has health benefits, hence it reduces stress and improves sleep quality.

On the other hand, negative or so-called “toxic” relationships lead people to the opposite side of the scale, it might lower one’s self-esteem and might higher the likeliness of mental health disorders. Lastly, socializing can lead one to have a stronger immune system.
As it was mentioned in the mental health section, positive social life can help reduce stress. If one can release stress, then it is unlikely to have chronic stress which can affect the hormone system as well as it may weaken one’s immune system.
Moreover, stress from negative or no social relationships may lead to poor/negative health habits such as alcohol abuse and passive lifestyle which may have a negative impact on the immune system as well.

On the other side, healthy social bonds can help develop positive health habits, for example sports activities, general motivation in everyday life and basic hygiene which improves body resistance.

2. Analysis

2.1 Requirements

The user stories below have been created with the participation of the customer. This was done in order to ensure a clear connection between what the customer wants and what the developers thinks the customer might want.

2.2 Functional Requirements

The list below is ordered by importance for the customer:

1. As a Viewer I want the ability to create an account, so I can participate in the community.

2. As a user I want the ability to authenticate myself as a user, so that I can have user privileges.
3. As a viewer I want the ability to see a list of dining events, so that I can see if I'm interested in the community.
4. As a user I want the ability to create an event, so I have the chance of dining with strangers.
5. As a user I want the ability to cancel an event, so that I can cancel in case of emergency.
6. As a user I want the ability to search for events, so I can see a selection of events matching my criteria.
7. As a user I want the ability to request to join a dining event that I find intriguing, in order to participate in dining events.
8. As a user I want the ability to leave an event that I have joined, so that the host doesn't think I'll show up if I make other plans.
9. As a user I want the ability to update my personal data, so I can keep it up to date.
10. As a user I want the ability to change my password, in order to keep my account safe.
11. As a user I want the ability to see all information about my account, so I can check if the information is correct.
12. As a user I want the ability to write a description for my event, so that I can inform other users about the event and its contents.
13. As a user I want the ability to see a map of where the dining event is being held, so that I can orientate myself.
14. As a user I want the ability to see a list of participants for a dining event, so I can see who I'm dining with.
15. As a user I want the ability to set a limit on how many users can join my event, so that I can plan accordingly.
16. As a user I want the ability to delete my account, so that I can leave the community.
17. As a user I want the ability to update event information, so that I can update it with new information.
18. As a user I want the ability to see a list of all past, present and future events that I'll be involved in, so that I can see what I've done for the community.
19. As a user I want the ability to get notifications when a request to join an event has been sent to me, so that I can stay on top of things.
20. As a user I want the ability to get notifications when a request to an event has been declined or accepted, so that I know if I can participate in the event or not.
21. As a user I want the ability to see a list of all unanswered requests, so that I can answer them when I get time.
22. As a user I want the ability to rate people that I've been dining with, so that I can let other people know what I thought of their company.
23. As a user I want the ability to upload a picture of myself, so that people can see what I look like.
24. As a user I want the ability to see other user's ratings, so that I can see if they're good company.

25. As an Admin I want the ability to ban users from the community, to keep the community free from trolls and toxic people.
26. As an Admin I want the ability to delete events, so that I can remove fake dining events from the service.
27. As an Admin I want the ability to see logs of what is happening in the system, so that I can see what is going on.

Constraints

1. The system must be coded in at least Java and C#.
2. The system must implement a 3-tier architecture.
3. The system must implement Sockets and webservices.

2.3 Non-Functional Requirements

There are no non-functional requirements.

2.4 Use Case Diagram

A use case is a list of event steps or actions between actors and a system in order to achieve different goals. The use case diagram below is based on the user stories. Inside the diagram there are three external entities as actors and many different use-cases to showcase how "Dining with strangers" intents to work (figure 1). The actors have been described as the following:

Viewer: Ability to either view dining events or create account.

User: Ability to create event, manage events, manage account, manage messages, manage requests and view history. The view history is made of previous actions based on the previously committed manage requests and manage events.

Admin: Ability to view user reports, administer users, view logs and administer events.

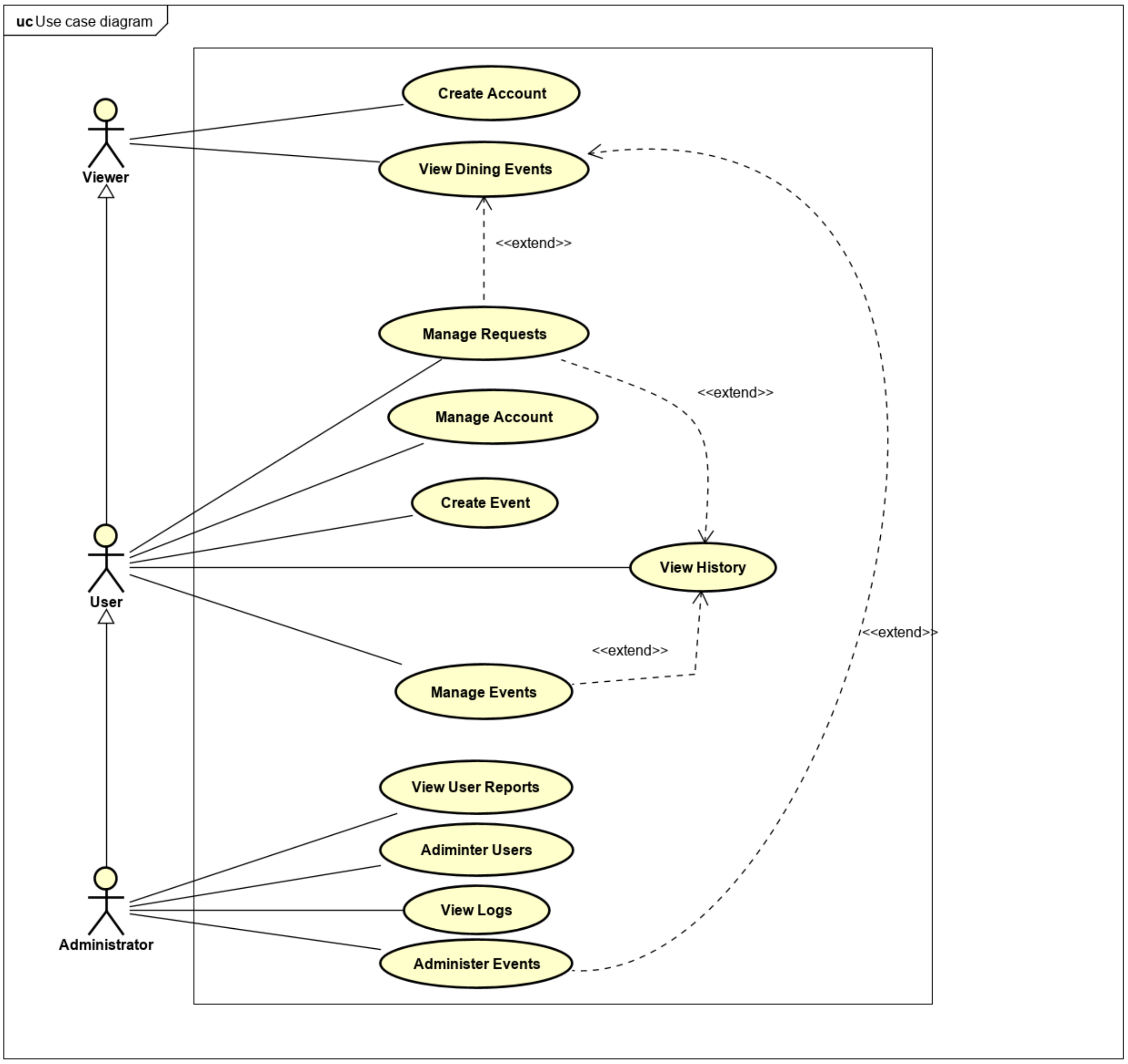


Figure 1 Use case diagram

2.5 Activity Diagram

Activity diagram is an important behavioral diagram in UML to describe dynamic aspects of the system. An activity diagram is used to show the steps and parallel processes of a use-case description. In order to create the example (figure 2), <<Create Event>> use-case description´s branch and exception sequences were followed. Hence, it shows all possible paths for executing the use case. In this case the activity diagram has two end scenarios based on the User´s actions. The remaining activity diagrams can be found in Appendix B: Activity diagrams.

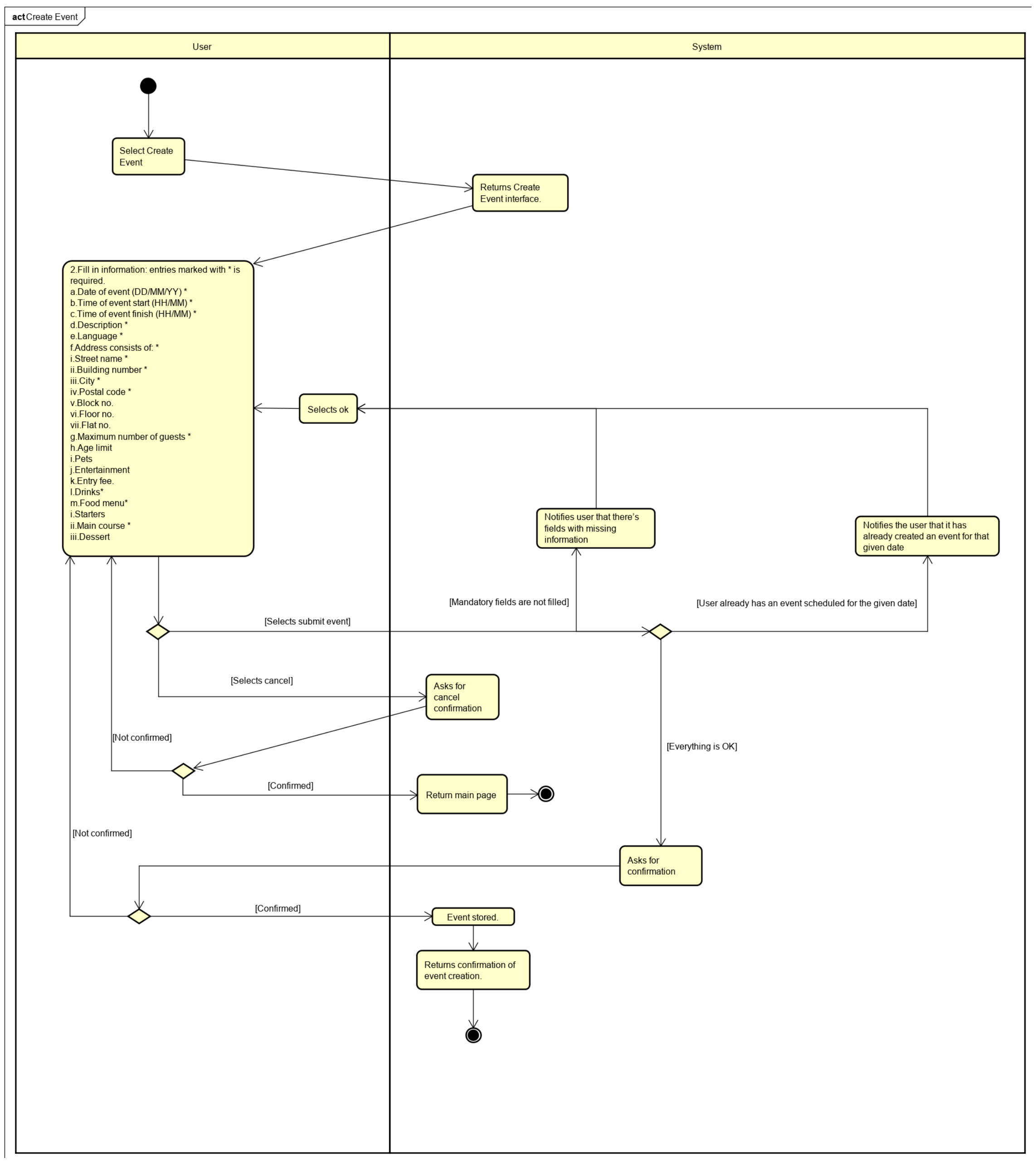

act Create Event
User
System
Select Create Event
Returns Create Event interface.
2.Fill in information: entries marked with * is required.
a.Date of event (DD/MM/YY) *
b.Time of event start (HH/MM) *
c.Time of event finish (HH/MM) *
d.Description *
e.Language *
f.Address consists of: *
i.Street name *
ii.Building number *
iii.City *
iv.Postal code *
v.Block no.
vi.Floor no.
vii.Flat no.
g.Maximum number of guests *
h.Age limit
i.Pets
j.Entertainment
k.Entry fee.
l.Drinks*
m.Food menu*
i.Starters
ii.Main course *
iii.Dessert
Selects ok
Notifies user that there's fields with missing information
Notifies the user that it has already created an event for that given date
[Mandatory fields are not filled]
[User already has an event scheduled for the given date]
[Selects submit event]
[Selects cancel]
Asks for cancel confirmation
[Not confirmed]
[Confirmed]
Return main page
[Everything is OK]
[Not confirmed]
Asks for confirmation
[Confirmed]
Event stored.
Returns confirmation of event creation.

Figure 2 Activity diagram - Create event

2.6 System sequence diagram

This figure displays interaction of the user with the system where all of the backend details and implementation are abstracted (figure 3).

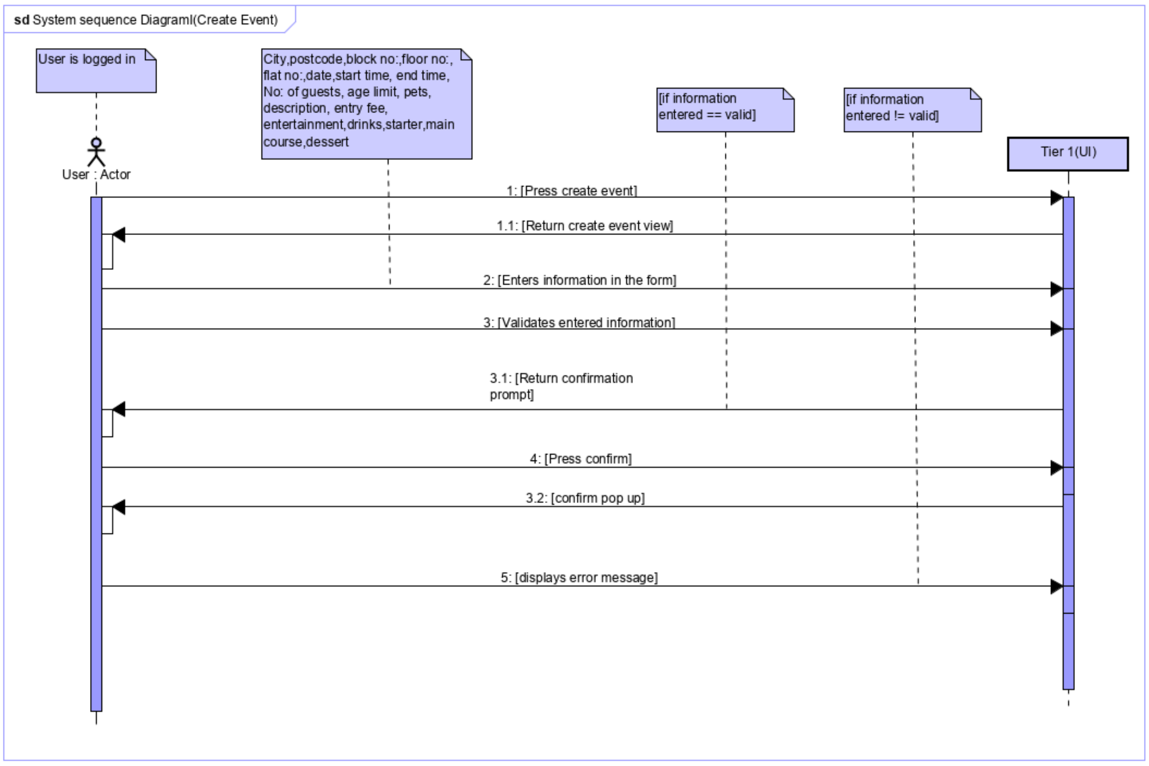

sd System sequence DiagramI(Create Event)
User is logged in
City,postcode,block no:,floor no:, flat no:,date,start time, end time, No: of guests, age limit, pets, description, entry fee, entertainment,drinks,starter,main course,dessert
[if information entered == valid]
[if information entered != valid]
Tier 1(UI)
User : Actor
1: [Press create event]
1.1: [Return create event view]
2: [Enters information in the form]
3: [Validates entered information]
3.1: [Return confirmation prompt]
4: [Press confirm]
3.2: [confirm pop up]
5: [displays error message]

Figure 3 Sequence diagram

2.7 Domain Model

The domain model is the outcome of various clarifications followed through analysis. Here, the relationships between classes are specified to carry forward the implementation later on. The User creates a dining event and other users can participate in a dining event by sending a request to join the dining event which has an address, and that in turn has a city. A User can also be an Admin to administer or manage other users and dining events. A user has an account history which includes history of requests sent to join a dining event. A viewer is a user, who can view events.

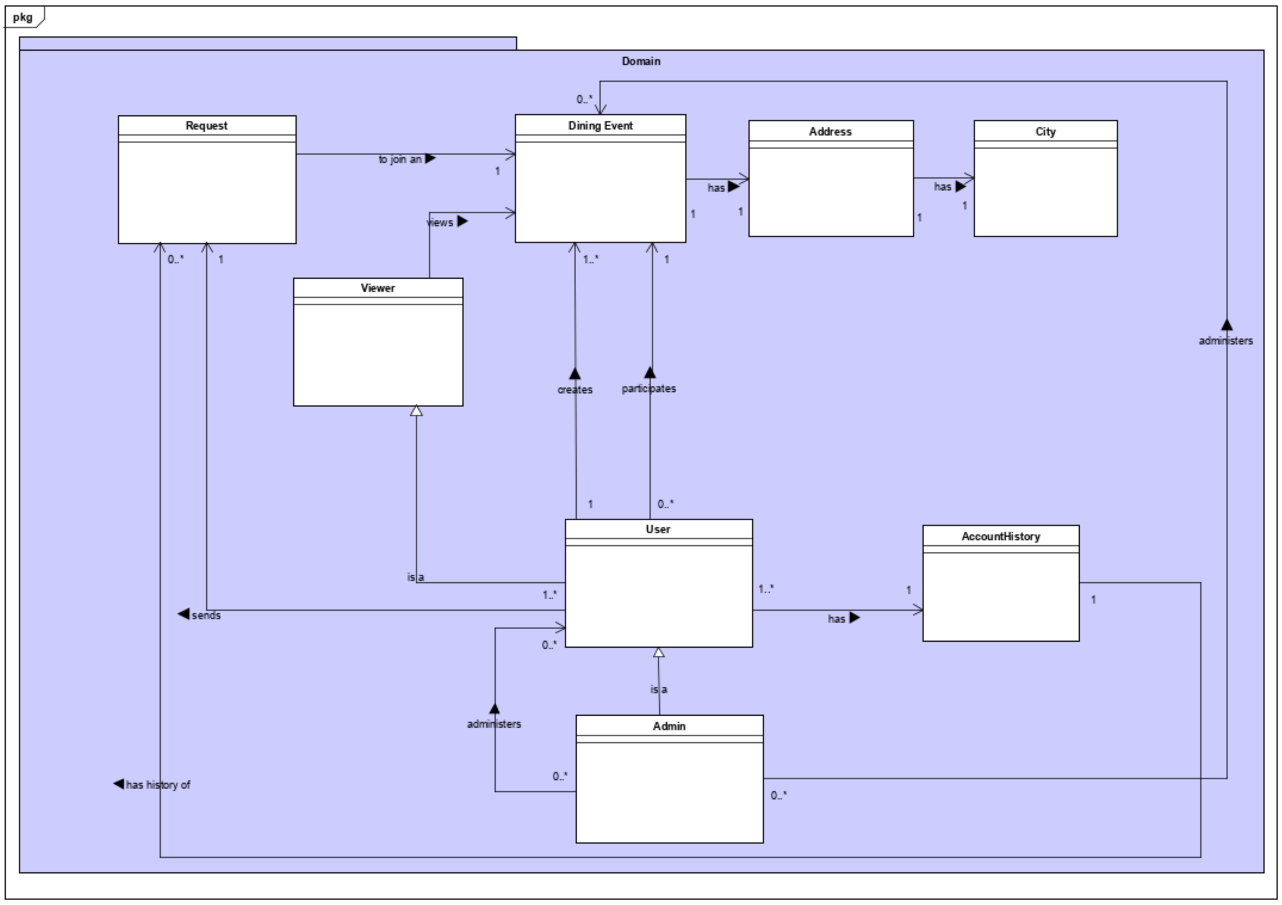


Figure 4 Domain model

2.8 Security

When working with software systems it is important to identify what kind risks or threats can impact the system in any kind of way. These possible attacks and threats have been identified by analyzing the use case descriptions in the different use cases.

There are three different types of threats when analyzing threats for a software system:

Intentional threats
Purposeful actions taking against the system with the goal of the attacker is to harm the system.

Unintentional threats
Considered to be human errors, often employees who does not mean to harm the system.

Environmental threats
Threats caused by non-human agents, it could be, however not limited to;
Natural disasters
Earthquakes

Flood
Fire
Animals and wildlife

After where and whom the threat is coming from, the goal of the attacker can be identified. The goal of the attacker is often referred to as STRIDE, which are different goals the attacker may look to achieve, they are;

Goals	**What to look at**
Spoofing identity	Authentication
Tampering	Integrity
Repudiation	Confirmation
Information disclosure	Confidentiality
Denial of service	Availability
Elevation of privileges	Authorization

The means of the attacker can be identified, now that the goal of the attacker has been found. There are two different means of attacking.

- **Active**
  - Eavesdropping
  - Traffic analysis
- **Passive**
  - Modification of message
  - Replay attack
  - Masquerade
  - Blocking

When the threat and its type has been identified, it is important to figure out where and by whom (EINOO) the attack is conducted from and by.

Who is separated into two categories:

- **E**xternal
- **I**nternal

Where is separated into three categories:

- **N**etwork
- **O**ffline
- **O**nline

The power of the attacker can be pin pointed, with the means of the attacker already distinguished. The power of the attacker can be the following:

Brute force
Every key is tried.

COA – Ciphertext-Only-Attack
The attacker knows nothing but the ciphertext of one or several messages encrypted with the same key.

KPA – Known Plaintext Attack
Attacker has the knowledge of paired plaintexts and ciphertexts encrypted with the same key.

Chosen-Plaintext Attack
The attacker can choose a plaintext and look at the paired ciphertext.

Chosen-ciphertext Attack
The attacker can choose a ciphertext and get decrypted plaintexts back.

Lastly where was the mistake, which is identified by TPM:

Threat identification was incomplete.

Policies did not work as intended or allowed things that should not have been allowed.

Mechanisms were circumvented.

With this knowledge it is possible to make a threat model and a threat assessment.

Threat type	Threat	Goal of the attacker: STRIDE → what to look at	Means of the attacker	Where and by whom: EINOO	The power of the attacker

Intentional	Phishing	Spoofing identity → Authentication
		
		Elevation of privileges → Authorization
			Masquerade(Active)
			Blocking(Active)
				External
				
				Online	

Intentional	MitM – Man-in-the-middle (IP Spoofing)	Spoofing identity → Authentication
		
		Elevation of privileges → Authorization
			Eavesdropping(Passive)
			Masquerade(Active)
				External
				Network	

Intentional	MitM (Session hijacking)	Spoofing identity → Authentication
		
		Denial of service → Availability
		
		Elevation of privileges → Authorization
		
		Tampering → Integrity
			Eavesdropping(Passive)
			Masquerade(Active)
			Blocking(Active)
			Modification of message(Active)
				External
				Network
					COA
					KPA

Intentional	MitM (Replay)	Spoofing identity → Authentication
		
		Elevation of privileges → Authorization
			Eavesdropping(Passive)
			Masquerade(Active)
			Replay(Active)
				External
				Network
					COA
					KPA

Intentional	TCP SYN flood attack	Denial of service → Availability		Online External	
Intentional	Ping of Death attack	Denial of service → Availability		Online External	

Intentional	Drive-by-Attack	Denial of Service → Availability
		
		Tampering → Integrity
		
		Elevation of privileges → Authorization
		
		Spoofing identity → Authentication
			Masquerade(Active)
			Blocking(Active)
				Online
				External	

Intentional	SQL Injection attack	Tampering → Integrity Information disclosure → Confidentiality		Online Network External Internal	

Intentional	Password attack	Spoofing identity → Authentication
		
		Elevation of privileges → Authorization
			Masquerade(Active)
			Eavesdropping(Passive)
				Online
				Network
				External
				Internal
					Brute force
					COA
					KPA

Intentional	Birthday attack	Information disclosure → Confidentiality Spoofing identity → Authentication Tampering → Integrity Elevation of privileges → Authorization	Masquerade (Active) Eavesdropping (Passive) Modification of message (Active)	Online External	Brute force

	Incident likelihood Threat frequency + Preventive measures.		Incident consequence Threat effect + Corrective measures.		Risk Incident likelihood + Incident consequence
Threat	Threat frequency	Preventive measures	Threat effect	Corrective measures	
Phising	Frequent	Critical thinking Sandboxing Analysis of email headers	Access to unauthorized information Could be connected to a malware attack		High

MitM (Session hijacking) (Spoofing) (Replay)	Rare	Key exchange (HTTPS) Public key pair-based authentication VPN Certificates Hash functions	Hijacker getting access to unauthorized information Denial of service for the targeted user.		Medium
TCP SYN Flood attack	Frequent	Firewall Connection queue size SSL/TLS	Service unavailable to other users	Use backup server	High
Ping of Death attack	Regular	Firewall checking fragmented IP packets for maximum size.	Denial of services for users.	Use server backup	low
SQL injection attack	Rare	Setup principle of least privilege (PoLP) on in database(s). Validation of user input in application layer	Access to unauthorized information		Low
Password Attack	Frequent	Strict password specifications Block user after multiple unsuccessful logins Multi factor authentication Captcha	Access to unauthorized information. Unauthorized access to account(s).	Make users change passwords. Account deletion	Medium

Birthday Attack	Rare	Digital signatures MACs	Integrity of message is lost. Unauthorized access.	Create new integrity mechanisms	low

3. Design

3.1 High level Design diagram

This part of design gives an overview of heterogeneous tiers (The choice of programming languages and services used) and their communication protocols between them as shown below:

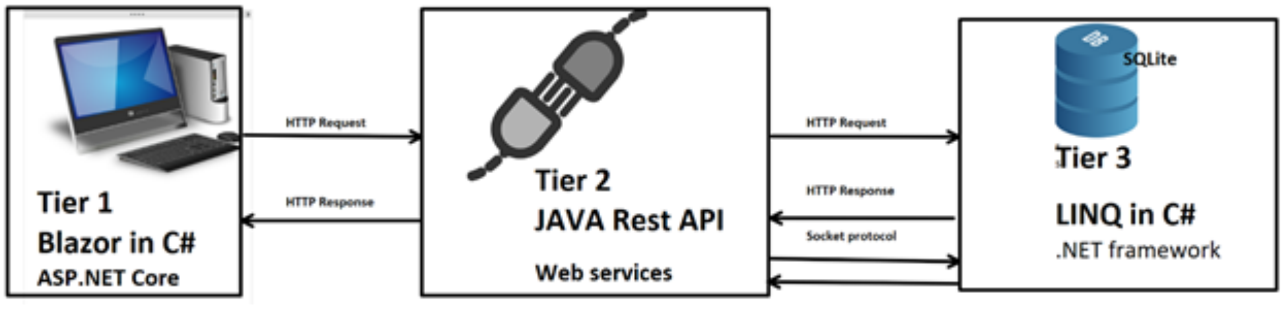


Figure 5 heterogeneous system architecture

3.2 UI Design – first tier

The user interface has been designed based on use case analysis. The fields required to complete action such as creating an event, user registering etc. are covered by the user stories. The user interaction and system response were designed to follow the scenarios from the use cases. Blazor Server was used to develop the client-side interface. The framework was chosen because of its interactivity. Moreover, the framework allows to develop applications written in C#, HTML and CSS without having to implement JavaScript code.

Dependency Injection is a design pattern used to develop loosely coupled code. The purpose of using Dependency Injection is to maintain the code and make it more flexible for later changes. The dependency can be injected and used in classes for example as a service. Objects, which use the dependencies can access all the features and methods defined in the dependency interface. It is an alternative approach where objects do not create instances of other objects they are using.

Due to the dependency injection of HTTP services, it was possible to develop first tier independently by creating substitute implementation for IUserInterface and IEventInterface. The role of those classes was to implement methods and return expected objects in order to render the desired view. Both ad hoc class and actual HTTP class methods return the same objects as a result.

Injection of the dummy service implementation in the CreateEvent.razor:

```
@inject JSLogin.Data.IEventService DummyEventService
/…
@code{
/…
void Create()
    {
      DummyEventService.PostNewEvent(newEvent);
       NavigationManager.NavigateTo("searchevent");
       toastService.ShowToast("The event has been created",
ToastLevel.Success);
    }
```

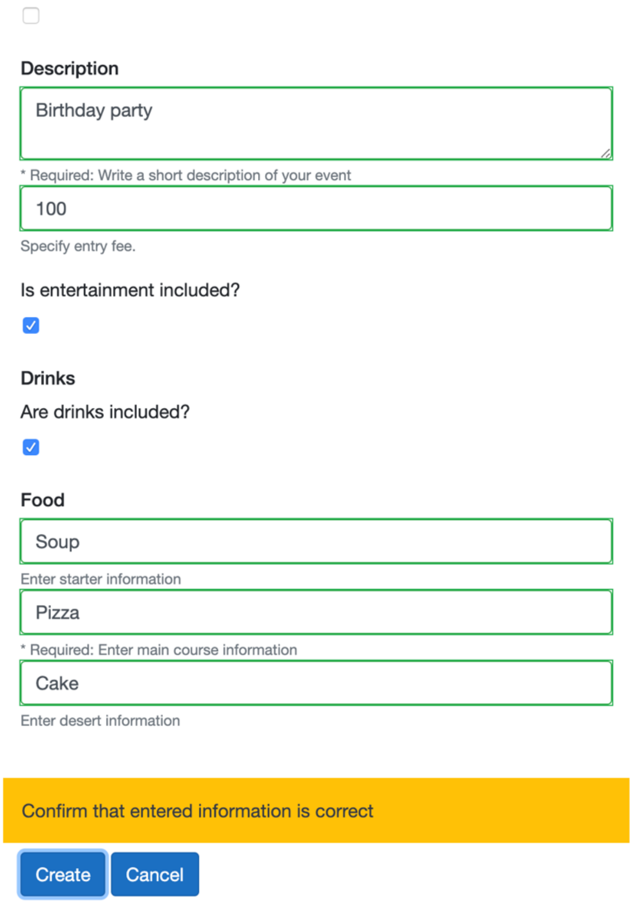
Service configuration in Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{
    /…
    services.AddScoped<IUserService, DummyUserService>();
    services.AddScoped<IEventService, DummyEventService>();
    /…
}
```

The project has been structured so that it contains following packages: Data, Pages, Services, Shared and wwwroot. The Data package holds representation of business model entities. Pages contain interface files such as CreateEvent.razor, EventDetails.razor etc. Services package holds service interfaces and their implementation, including Http and notification services. The Shared folder contains .razor files as well as Razor pages which are used and dependant on the main layout. Folder wwwroot holds static files. The files in the folder define styling of the layout, and JavaScript functions. The files contained in the folder are web addressable.

Following views has been implemented and tested, focused on data rendering and user interaction with the system: AccountSettings.razor, Register.razor, EditEvent.razor, EventDetails.razor, EventHistory.razor, History.razor, SearchHistory.razor. User interaction and inputs are based on use case scenarios and user stories (see Appendix I: User manual).

In the CreateEvent.razor (figure 5) in order to create an event, user has to pass following parameters: City, Post code, Street, Block number, Floor number, Flat number, Date, Start time, End time, Number of guests, Age limit, Pets, Description, Entry fee, Entertainment, Drink, Starter, Main course, Dessert. Success use case scenario assumes that after filling out the form, user has to press Create button. System responds with confirmation prompt. The event creation happens when user confirms by pressing Create button. The UserService dependency is calling HTTP POST method.


☐

Description

Birthday party

* Required: Write a short description of your event

100

Specify entry fee.

Is entertainment included?

☑

Drinks

Are drinks included?

☑

Food

Soup

Enter starter information

Pizza

* Required: Enter main course information

Cake

Enter desert information

Confirm that entered information is correct

Figure 6 Create events interface

3.3 Rest Java API - second tier

Second tier was designed with a few controllers like EventController, LoginController,RequestController and UserController extending the ControllerNeeds class. To view a more elaborate design of the class diagram it is appended to Appendix E: Class diagrams

3.3.1 Sequence diagram

The sequence diagram gives a more detailed design of how a method is called throughout the tiers to carry out a user's action in a system.
Following figure presents sequence diagram for creating an event.

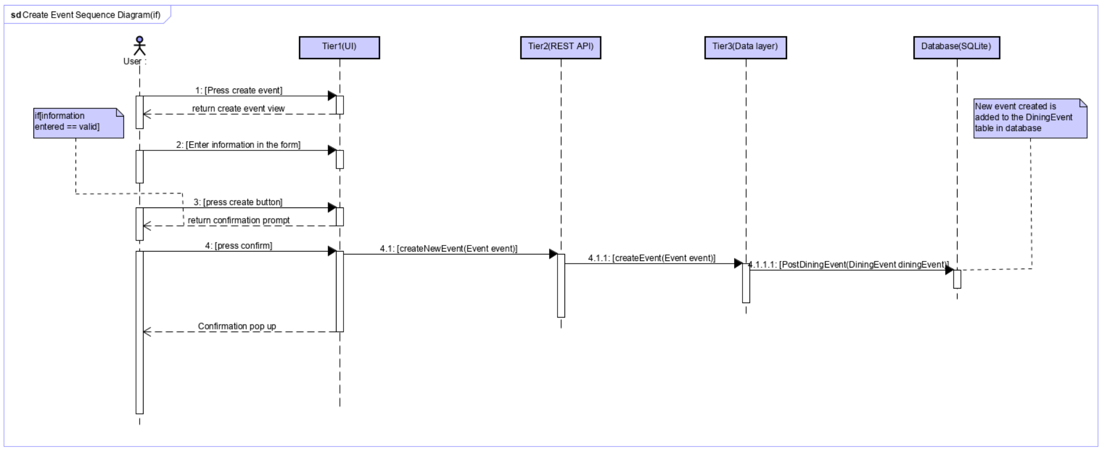

sd Create Event Sequence Diagram(if)
User :
Tier1(UI)
Tier2(REST API)
Tier3(Data layer)
Database(SQLite)
1: [Press create event]
return create event view
if[information entered == valid]
New event created is added to the DiningEvent table in database
2: [Enter information in the form]
3: [press create button]
return confirmation prompt
4: [press confirm]
4.1: [createNewEvent(Event event)]
4.1.1: [createEvent(Event event)]
4.1.1.1: [PostDiningEvent(DiningEvent diningEvent)]
Confirmation pop up

Figure 7 Create event sequence diagram

3.3.2 Dependency injection in second tier

Dependencies which can be used by an object is known as a service and the injection is literally passing of the dependency to another object that uses it. Hence, in the second tier the controllers have no direct dependencies with the other classes because of this technique. The chosen framework to work with this technique is Spring framework in Java, where it has a DI injection container which is responsible for the creation (with factory design pattern) and wiring together these objects with dependencies. These dependencies manage components of the tier and these objects are called Spring Beans (Appendix E: Class diagrams).

3.3.3 Socket application layer protocol

The socket protocol was designed to be used between the 2nd and 3rd tier for login process with a few response statements as clearly stated in the diagram and later explained in implementation (figure 6).

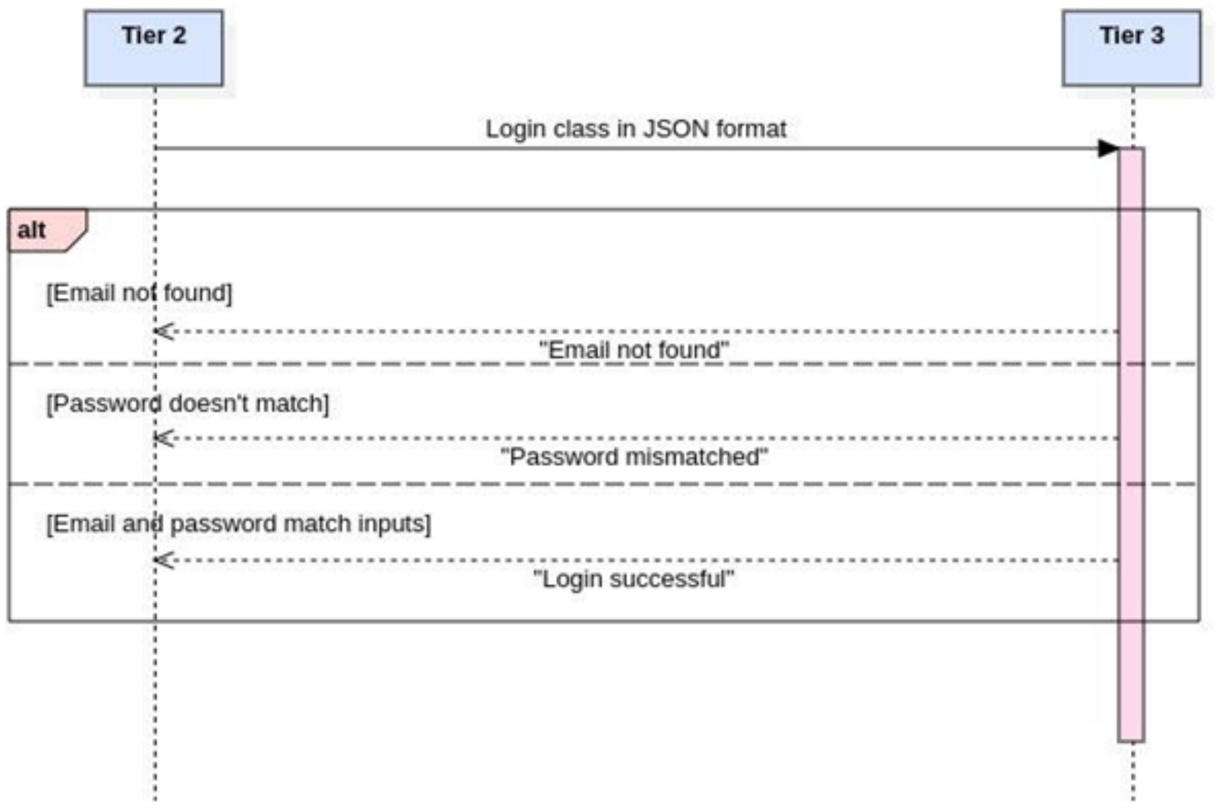


Figure 8 Socket application layer protocol (log in)

3.4 Data layer – third tier

3.4.1 ER Diagram

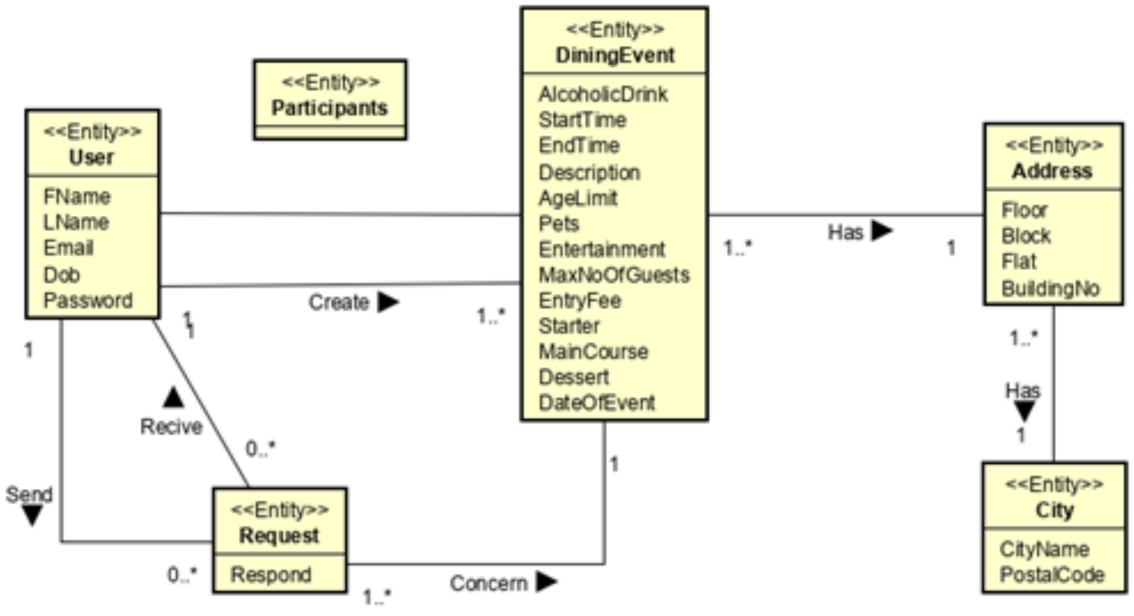
ER diagrams were designed based on conceptual, logical and physical models. All diagrams contain relations and entities, but they differ in their purposes. A general understanding for the three models is the following: Business analysis use conceptual and logical model for modeling the data derived from the user stories, on the other hand physical model is used for creating the design of the database. In order to create ERD user stories were analyzed. After the analysis the following entities were defined: User, DiningEvent, Address, City, Requests, Participants. The connections and relations are the following:

1. A user is able to create many dining events.
2. A user is able to send and receive many requests corresponding for a dining event.
3. Participants is defined as a relation between users and dining events, in order to keep track of the number of users participating in each dining event.
4. One dining event has an Address and each address can map to many dining events.

5. Many addresses may have the same city.

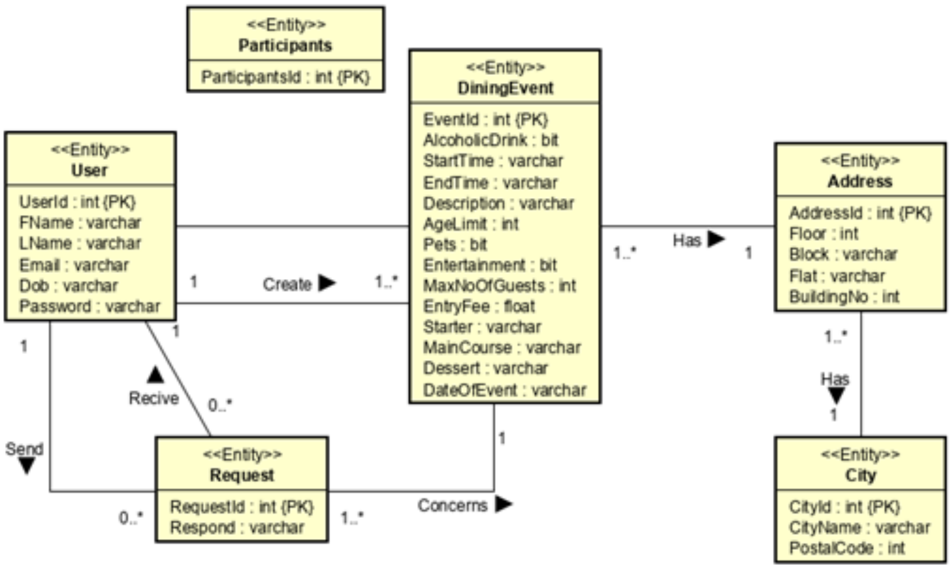
3.4.2 Conceptual ER Diagram

Then entities and relations were defined based on Dining with stranger´s needs. Conceptual ERD is used for showcasing the entities, their attributes and their relations. The conceptual ER diagram is not detailed enough to design a database, since it is the simplest model among the relational database design models.



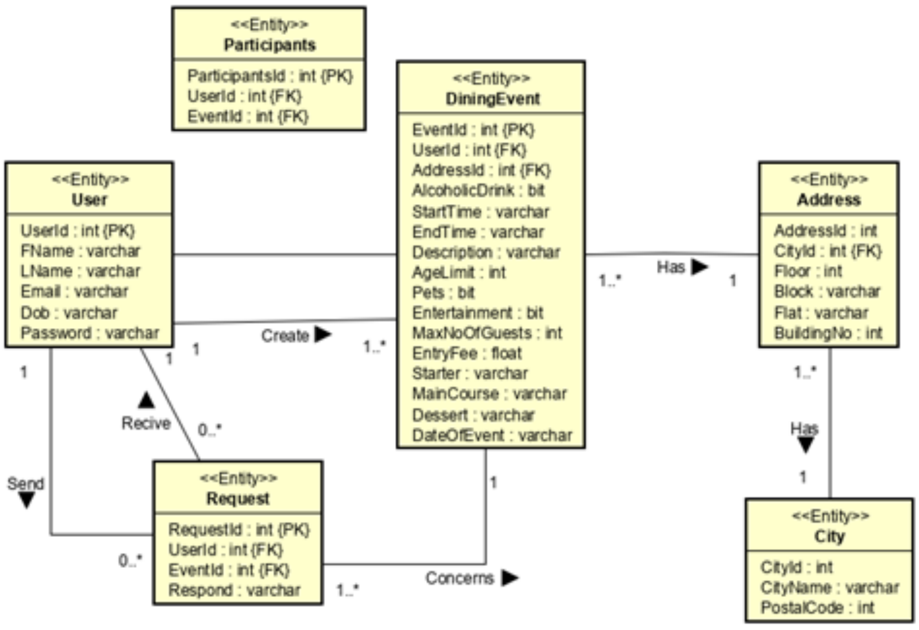
3.4.3 Logical ER Diagram

It represents a more complex model with attribute types and primary keys.



3.4.3 Physical ER Diagram

Physical ERD represents the actual design of the implemented database. It has been created based on the logical ERD since, Physical ERD translates the relations from the logical diagram into a model which can be implemented into the database. Here all the attributes, relations, primary and foreign key are visible.



3.4.4 Normalization

Normalization is the process to reduce the data dependency and improve data integrity by building a relational database according to series of normal forms. According to the ERD normalization were followed till the third form. To understand the normal forms the forms were defined the following way:

1. After using the 1NF each column must contain single values and each record have to be unique.
2. In order to reach 2NF tables must be in 1NF. Primary Key must be added to 1NF to uniquely identified the records/entities.
3. 2NF has to be applied on the entities. Where it is necessary entities must be divided to prevent transitive functional dependencies. Foreign keys must be defined, and tables has to be separated if it's necessary.

Dining with stranger’s database has been designed based on normalization as it was previously mentioned.

3.5 Security design

It would make sense to investigate implementing the following mechanisms, based on the security threats analyzed in analysis.

Sandboxing
Sandboxing is an isolated environment where suspicious code or links can be executed. This means that sandbox proactively helps detecting phishing or malware by executing it in a safe environment where their behavior and output can be observed without harming the system.

VPN
A virtual private network is used to establish a secure, encrypted connection between a remote user and the system network. In order to setup a VPN tunnel, one of the following three options could be chosen:

IPSec
IPSec can be implemented in two ways. The transport mode can be implemented, thus only the data portion of a packet will be encrypted, and the header left unencrypted. The second option is the tunnel mode, where the entire packet is encrypted.
For this protocol to work, both the sending and receiving devices must share a public key.
IPSec provides confidentiality, authentication and integrity

PPP – Point-to-point tunneling protocol
Is a data link layer protocol that establishes a direct connection between sender and receiver without any other networking in-between. To achieve authentication either a password authentication protocol or a challenge handshake protocol could be implemented.

L2TP – Layer two tunneling protocol
A L2TP connection is made up of a tunnel and a session. The tunnel’s purpose is to provide transport between two endpoints, and it only carries control packets.

The session is contained within a tunnel and caries user data between the endpoints. A tunnel can support multiple sessions.
L2TPs often used in conjunction with IPSec. This is duo to L2TP not providing any confidentiality or authentication by itself. Furthermore, it is not uncommon to see PPP carry sessions within a L2TP tunnel.

Certificates
Are a way to prove ownership of public keys. Certificates are issued by certificate authorities.

Hash functions
Hash functions maps data to fixed-size values, these fixed-size values are called hash values. One of the uses of hash functions could be the hashing of passwords in the database to avoid storing sensitive information in plain text.

Firewall
Firewalls are used as a barrier between trusted internal networks and untrusted external networks. The firewall works by monitoring and controlling network traffic. The firewall has a predetermined set of security rules that it then must monitor and control. For example, checking fragmented IP packets for maximum size to avoid pings of death.

PoLP
Principle of least privileges is the practice of limiting user access to a point where they are able to perform their tasks, and no more. Then the system can have superusers such as admins that have unlimited access to the system.

Multi factor authentication
Ensures that users cannot gain access to a system without presenting two or more factors to authentication, thus enhancing the authentication strength.

SSL/TLS
TLS is a cryptographic protocol which provides security of information over networks. TLS is intended to mitigate eavesdropping, tampering and message forgery. TLS is often implemented with HTTPS.

MACs
Message authentication codes are used to protect the integrity of the data as well as authenticity. This is done by equipping a message with a small piece of information that authenticates the sender. This way senders cannot refuse that they sent the message which in turn eliminates repudiation.

4 Implementation

4.1 UI Implementation – first tier

4.1.1 Introduction to ASP.NET Core Blazor

The UI for the system was developed using ASP.NET Core Blazor framework. The framework allows user interaction by implementing C# code and rendering with the use of HTML and CSS. The Blazor components are in the form of Razor markup page. The components have a .razor file extension and combine HTML and C# code which allows designing UI composition and more flexible rendering of the web page. The system implements Blazor Server which is currently supported in ASP.NET Core 3.0. The components use representation of Document Object Model. The content of the page is compared during the runtime, only the content which has been changed is applied.

4.1.2 Data and models

Data package contains models and services. The models are used to represent system entities such as Event or User. They are similar to the entities contained in database, they only have attributes which are necessary for UI rendering and data logic. The following code implements User object which is used among others for rendering Account Settings.

```
using System;
using System.ComponentModel.DataAnnotations;
using System.Text.Json;
using System.Text.Json.Serialization;


    public class User
    {
        [Required(ErrorMessage = "Required")]
        [JsonPropertyName("id")]
        [Display(Name = "User Id")]
        public string UserId { get; set; }
        [Required(ErrorMessage = "Required")]

        [JsonPropertyName("fName")]
        [Display(Name = "First name")]
        [StringLength(15, ErrorMessage = "Name is too long.")]
        public string FirstName { get; set; }

    }
```

Attribute tags have been additionally added to the attributes in the User class. The Data Annotation feature can validate the user’s input or specify how data is displayed for the user in the interface. The *Required* tag indicates members of the class which has to be provided by the user. The *Display Name* tag allows to modify property's name which will be visible on an error message. The *JsonPropertyName* enables to control how JSON is serialized. The key will have the C# property's name by default. This tag enables to customize key name in JSON's key-value pairs.

4.1.3 UI components and rendering pages

The **Pages** package contains set of pages which are displayed in the browser when the app is running. Logged in user can choose to search for events, edit user settings, see history of events etc. by selecting specific tab in the navigation menu. The files have a .razor file extension. The request for specific page in the browser is specified by the @page directive. The below code is a partial implementation of following user story:

As a user I want the ability to create an account, so I can participate in the community.

```
@page "/register"
@using Data
@using Microsoft.AspNetCore.Blazor.Services
@using System.Threading.Tasks
@inject DiningApp.Data.IUserService UserService
@inject NavigationManager NavigationManager

/…
    <EditForm Model="@newUser" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText placeholder = "First name" @bind-Value="newUser.FirstName" />

/…
   <button @onclick="RegisterUser">Confirm registration</button>
   </EditForm>

@code {

private User newUser = new User();
/…

async Task HandleValidSubmit(){
 await RegisterUser();
}

async Task RegisterUser(){
 Console.WriteLine("[BUTTON] CONFIRM REGISTRATION”);
 await UserService.PostNewUser(newUser);
 NavigationManager.NavigateTo("success");
 toastService.ShowToast("You have successfully registered.",ToastLevel.Success);

}
}
```

The C# attributes and methods are defined in the *@code* block. The methods are used for event handling. The above example of registering a user, creates a field newUser. The newUser field is

assigned to the Model attribute in the Blazor's *EditForm* component. This component will render user registration form in the UI. The *InputText* elements define necessary fields which have to be filled out for successful registration. In the scope of input field, a *@bind-Value* attribute is defined. This function is binding user's input to the specific attribute in User class. The form is validated before firing the registering event. The fields would be validated in the *HandleValidSubmit()* method based on data annotations from the *User.cs* class which define validation logic.
The *RegisterUser()* method is called after clicking Confirm registration button. The dependency injection of UserService and NavigationManager allows following actions; calling *PostNewUser()* method in the UserService, which performs HTTP post request to register new user, followed by calling the *NavigateTo()* method, which will redirect user to default page.

4.1.4 Routing

The routing is done with the use of *@page* directive at the top of the file. Alternatively, router can use additional routes and use parameters. In this example two directives are implemented. In order to view event details, user has to select Search Events tab from the navigation menu. This will re-direct to */searchevents* page where it is possible to select specific event and view its details. By clicking the event, the specific event id is routed and redirected to */eventdetails* page.

```
<a class="selectablelink" href ="/eventdetails/@newEvent.EventID">
```

On the */eventdetails* page, second directive uses *eventID* parameter initialized in the *@code* block. The *eventID* of selected event is assigned to the *EventID* property.

```
@page "/eventdetails"
@page "/eventdetails/{eventID}"
@if(EventID == null)
{
  <a href="/searchevents">Go back to search events</a>
}

<h1 class="display-4">Event details @EventID </h1>
@code {

    Event details;
    [Parameter]
    public string EventID {get;set;}

    protected override async Task OnInitializedAsync() {

       details = await EventService.GetEventDetailsByID(EventID);
}
}
```

Field annotated as [Parameter] can be used outside of *@code* scope. For example, in the HTML section is possible to use parameter by adding @. Injected *EventService* allows calling methods in the class. The *OnInitializedAsync()* method will render the tree based on the parameters. The method will only fire once at the loading of page. Method will not fire again after any of the components is re-rendered. In the method, an event is fetched by calling *GetEventDetailsByID()* and passing *EventID* parameter and later assigned to the details field.

4.1.5 HTTP Communication and processing JSON information

The Blazor app issues requests to Java REST service. The app implements REST client to retrieve or post information to specific endpoint. The endpoints are specified in the Java Web API.
The implementation of *IUserService* and *IEventService* interfaces contains methods sending requests to Java server. *EventService* class implements methods specific to Event class such as *CreateEvent(), CancelEvent()* etc. The *UserService* handles events related to User class such as registering, editing account settings etc. The *HttpClient* class provides means to request and receive response from given URL specified by the Java server in 2nd tier. A POST method in the *EventService()* creates a *HttpClient* instance. The endpoint URL is specified as a *requestUrl*. Method delegates the task of processing the information, converting it into JSON format and posting data asynchronously to the Web API. Following code implements posting method, *CreateNewEvent()* which is used for posting newly created events in the system.

```
public async Task<string> CreateNewEvent(Event newEvent)
{
    Console.WriteLine("HTTP REQUEST: CREATE EVENT");
    Console.WriteLine(newEvent.ToString());

    HttpClient client = new HttpClient();
    string requestUrl = $"http://{IP}:8080/startApplication/api/newEvent";
    HttpResponseMessage response =
        await HelperMethods.PostAsJsonAsync(client, requestUrl, newEvent);
    return response.ToString();
}
```

An extension method is used to serialise data to JSON string:

```
public static async Task<HttpResponseMessage> PostAsJsonAsync<TModel>(this
HttpClient client, string requetsUrl, TModel model)
  {

var json = System.Text.Json.JsonSerializer.Serialize(model);
var stringContent = new StringContent(json, Encoding.UTF8, "application/json");
return await client.PostAsync(requetsUrl, stringContent);

   }
```

HTTP requests are used for fetching data from 2nd and 3rd tier. The following example implements a method for fetching list of events. This method is used in the Search events view. When the page is loaded, initially an *OnInintialisedAsyn()* method is called which renders a list of all of the available events stored in the database. User can specify the name of the city in search bar. This action will fire another HTTP request, which returns list of events at the city specified by the user. The following method implements search request where the city is specified as a user and taken as an argument.

```
public async Task<List<Event>> SearchEventsAsync(string city)
{
Console.WriteLine("HTTP REQUEST: SEARCH EVENT");

HttpClient client = new HttpClient();
string requestUrl = $"http://{IP}:8080/startApplication/api/search?city={city}";

string json = await client.GetStringAsync(requestUrl);
Console.WriteLine(json);
var result = JsonConvert.DeserializeObject <List<Event>>(json);
Console.WriteLine(result.ToString());

return result;

}
```

The JSON string is returned from the Java server, then it is deserialized into a list of Events and returned within the method.

HttpClient supports asynchronous methods, therefore *System.Threading.Tasks* namespace is used to handle concurrent methods. *UserService* and *EventService* methods are called directly from the view as a result of dependency injection of services. When calling these methods from the view, they also have to be encapsulated in the async Task method and await the result. Following code implements *SearchEvents.razor* class. The *EventService* is injected at the top of the file, in the @code block, Create method is executed when the Confirm registration button is pressed.

```
@page "/createevent"
@using System.Threading.Tasks
@inject DiningApp.Data.IEventService EventService
/…
async Task Create()
{
    await EventService.CreateNewEvent(newEvent);
    /…
    NavigationManager.NavigateTo("searchevent");
    toastService.ShowToast("The event has been created", ToastLevel.Success);
}
```

4.1.6 Notifications

The application uses notification system to inform the user, whether the registration, event creation or updates have been executed successfully. The component used for displaying notifications is Blazor Toast. It is reacting to the events invoked by a *ToastService*, registered in the *Startup.cs* class. The *ToastService* is injected in every page which need notification support.

```
@inject ToastService toastService
```

The *ShowToast()* method in the *ToastService* will display the message when the function is fired. *ToastService* implements *IDisposable* interface which will release any unused resources (IDisposable Interface (System), 2019). The following code implements calling *ToastService* registration cancellation method in the Register view.

```
void CancelRegistration()
{
    NavigationManager.NavigateTo("searchevents");
    toastService.ShowToast("You have NOT registered.", ToastLevel.Error);
}
```

4.1.7 Services configuration

Services used in the project are set up in the *ConfigureServices()* method in the *Startup.cs* file. This enabled dependency injection of specific services throughout the project. The components registered in this method are reusable. Services registered in the method include HTTP communication services for User and Event requests, notification services, authentication and cookie policies, configurations for *HttpContext* services.

```
 public void ConfigureServices(IServiceCollection services)
        {
           /…

          services.AddRazorPages();
          services.AddServerSideBlazor();
          services.AddScoped<IUserService, UserService>();
          services.AddScoped<IEventService, EventService>();
          services.AddScoped<ToastService>();
          services.AddHttpContextAccessor();


           /…
        }
```

The Scoped instances are created once per HTTP client request (Dependency injection in ASP.NET Core, 2019). The *AddScoped()* method will create an insane of a scoped service per each request and use it other calls within the scope of the request.

4.1.8 Authorization and access control

The authorization and authentication are used for configuring and managing security and access control throughout the application. The authorization checks determine the scope of accessibility of user interface for the user and the access rules and claims for different users (logged in users or visitors) for areas of the application. The authorization follows the implementation based on “Exploring authentication in Blazor” tutorial (Walker, 2019). This solution uses JavaScript to create a login and logout requester mechanism. The JavaScript component is reusable throughout the Blazor application. When the user is logged in, specific claims are given which allow the user to access bigger scope of the application, such as a possibility to create an event, view the history of events and edit or cancel any of them. In comparison, a user who is not logged in - a visitor can only access search events feature.

4.1.9 Authentication

The razor components can be hidden from the user based on being logged in.
Policies are configured in the *Startup.cs* file in the *ConfigureServices()* method.

```
services.AddAuthorization(options =>
{
    options.AddPolicy("MustBeLoggedIn", p =>
p.RequireAuthenticatedUser().RequireClaim("Role", "loggedin"));

});
```

```
The claim is given when the user successfully logs into the system from the
LoginModel:
public async Task<IActionResult> OnPostAsync(string username, string password)
{
 /…

    var claims = new List<Claim>{
        new Claim(ClaimTypes.Name, username),
        new Claim("Role", "loggedin")
    };

    var identity = new ClaimsIdentity(claims,
CookieAuthenticationDefaults.AuthenticationScheme);
    var principal = new ClaimsPrincipal(identity);
    await HttpContext.SignInAsync(
        CookieAuthenticationDefaults.AuthenticationScheme, principal);

    return LocalRedirect(Url.Content("~/"));
}
```

In order to access the *CreateEvent.razor* page, the user must be granted a “loggedin” claim. That is done by using [Authorize] attribute which sets the scope of pages visible for logged in user versus visitor.

```
@page "/createevent"
@attribute [Authorize(Policy = "MustBeLoggedIn") ]
```

The policy-based authorization is also used in the navigation menu, which appears on the left side of the user interface. The link to the page is encapsulated in the <Authorized> tag and a specific policy is used to determine the visible scope.

```
 <AuthorizeView Policy="MustBeLoggedIn">
    <Authorized >
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="createevent">
                <span class="oi oi-plus" aria-hidden="true"></span> Create Event
            </NavLink>
        </li>
    </Authorized>
</AuthorizeView>
```

In order to specify the view for a visitor, a <NotAuthorized> tag is used. This would enable the visitors to access the Search events interface without having to create an account.

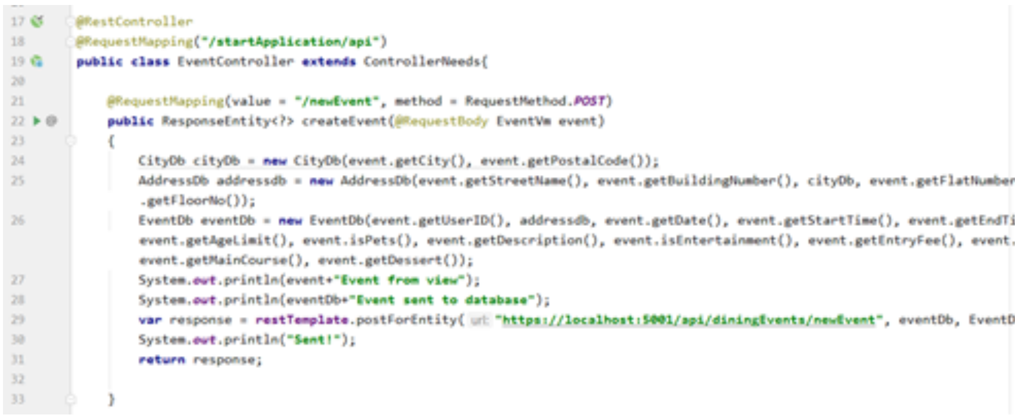
```
<AuthorizeView Policy="MustBeLoggedIn">
    <NotAuthorized >
<li class="nav-item px-3">
  <NavLink class="nav-link" href="searchevents">
  <span class="oi oi-magnifying-glass" aria-hidden="true"></span> Search
Events
</li>
    </NotAuthorized>
</AuthorizeView>
```

4.2 REST Java API - second tier

The implementation part of the system gives an elaborate view of how the Analysis and design was transitioned into code to implement its functional requirements.

The second tier was developed in Java Spring framework using Spring Boot to act as an API between the first tier and third tier to be able to handle and translate information added or retrieved. At the end of Analysis, the domain model was split into two in order to translate the serialized data passed between the frontend and backend. In Spring's structure for building a RESTful web service, a controller handles and routes the HTTP requests with specific annotations.

The following code snippets from Event controller class explains how a controller, HTTP routing and annotations work:



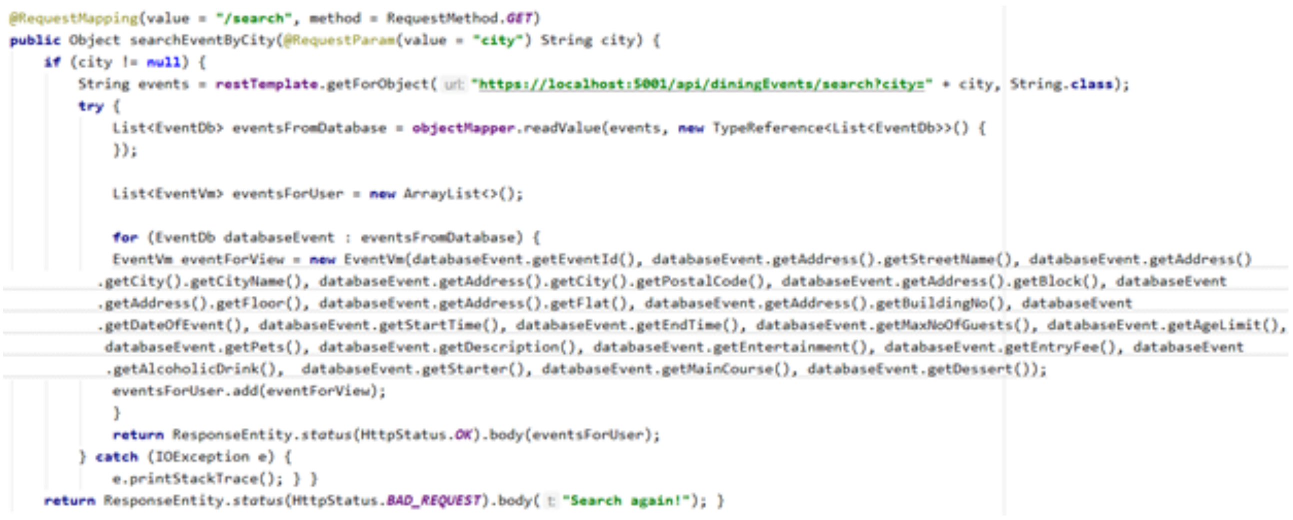
```java
@RestController
@RequestMapping("/startApplication/api")
public class EventController extends ControllerNeeds{

    @RequestMapping(value = "/newEvent", method = RequestMethod.POST)
    public ResponseEntity<?> createEvent(@RequestBody EventVm event)
    {
        CityDb cityDb = new CityDb(event.getCity(), event.getPostalCode());
        AddressDb addressdb = new AddressDb(event.getStreetName(), event.getBuildingNumber(), cityDb, event.getFlatNumber(), event.getBlockNo(), event
        .getFloorNo());
        EventDb eventDb = new EventDb(event.getUserID(), addressdb, event.getDate(), event.getStartTime(), event.getEndTime(), event.getMaxNoOfGuests(),
        event.getAgeLimit(), event.isPets(), event.getDescription(), event.isEntertainment(), event.getEntryFee(), event.isDrinksVm(), event.getStarter(),
        event.getMainCourse(), event.getDessert());
        System.out.println(event+"Event from view");
        System.out.println(eventDb+"Event sent to database");
        var response = restTemplate.postForEntity( url: "https://localhost:5001/api/diningEvents/newEvent", eventDb, EventDb.class);
        System.out.println("Sent!");
        return response;

    }
```

Annotations:

- *@RestController* is mentioned above the Event controller class to specify it is a HTTP request handler.
- *@RequestMapping* is mentioned above the Event controller class and *createEvent()* method to delegate requests with a specific URL path to a controller and method.

The *createEvent()* method is a HTTP POST method which receives an event object from the frontend. This method extracts city, address and event objects to post it to the data layer as its model is designed differently compared to the frontend. After the event object is extracted to be sent to Data layer in the form of JSON which was marshalled by Jackson - a Java spring object serializer and de-serializer, *RestTemplate* is used to get access to a range of operations where it acts as a client to the Data layer. In the example above, "*postForEntity(URL,Object,classname.Class)"* is used to specify the URL to a HTTP post method in the data layer that adds the event to the database. The method then returns a *ResponseEntity* that represents the whole HTTP response with either a status code, header or body.

The following code snippet for *searchEventByCity(String city)* from the Event controller which specifies more details on HTTP and annotations in spring:



```
@RequestMapping(value = "/search", method = RequestMethod.GET)
public Object searchEventByCity(@RequestParam(value = "city") String city) {
    if (city != null) {
        String events = restTemplate.getForObject( url: "https://localhost:5001/api/diningEvents/search?city=" + city, String.class);
        try {
            List<EventDb> eventsFromDatabase = objectMapper.readValue(events, new TypeReference<List<EventDb>>() {
            });

            List<EventVm> eventsForUser = new ArrayList<>();

            for (EventDb databaseEvent : eventsFromDatabase) {
            EventVm eventForView = new EventVm(databaseEvent.getEventId(), databaseEvent.getAddress().getStreetName(), databaseEvent.getAddress()
           .getCity().getCityName(), databaseEvent.getAddress().getCity().getPostalCode(), databaseEvent.getAddress().getBlock(), databaseEvent
           .getAddress().getFloor(), databaseEvent.getAddress().getFlat(), databaseEvent.getAddress().getBuildingNo(), databaseEvent
           .getDateOfEvent(), databaseEvent.getStartTime(), databaseEvent.getEndTime(), databaseEvent.getMaxNoOfGuests(), databaseEvent.getAgeLimit(),
            databaseEvent.getPets(), databaseEvent.getDescription(), databaseEvent.getEntertainment(), databaseEvent.getEntryFee(), databaseEvent
            .getAlcoholicDrink(),  databaseEvent.getStarter(), databaseEvent.getMainCourse(), databaseEvent.getDessert());
            eventsForUser.add(eventForView);
            }
            return ResponseEntity.status(HttpStatus.OK).body(eventsForUser);
        } catch (IOException e) {
            e.printStackTrace(); } }
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body( t: "Search again!"); }
```

Annotations:

- *@RequestParam* is used to bind a web request parameter to a HTTP method request and the value represents the name of the placeholder for the parameter.

The *searchEventByCity(@RequestParam(value = "city"))* is a HTTP GET method which receives a parameter city of type String from the frontend, which is bonded to the method variable and is routed to the data layer using *RestTemplate* object. The list of events received from the data layer after the search is then un-marshalled by Jackson and converted to java object and then vice versa to be sent to the frontend, to be viewed with a *ResponseEntity* status code "OK". In another scenario, if the city variable is null or not found in the database, the *ResponseEntity* status code returned to a view will be "Bad Request".

4.2.1 Socket application layer protocol Implementation

Socket connection was implemented on both second and third tier to send login credentials with a communication design protocol mentioned in the Design section.

Implementation in Java as client can be found in Appendix F: Java sockets implementation.

In the login controller class, *@RequestMapping* routes the method login to the data layer login method to connect to the server socket. In this method a socket is made to communicate over *InputStream()* and *OutputStream()* and when sending JSON over the stream it is then converted to bytes in order to be more compatible on the receiving end in the data layer. If the email which is unique has passed through a successful scenario as mentioned in the design section, it will return a String message "Login successful", otherwise "Email not found" or "Password mismatched" with a "Bad Request" ResponseEntity.

Implementation in C# as a listener can be found in Appendix G: C# listener.

The *SocketLogin()* method is defined and called inside the *EventDbContext* class, a socket is created to listen or accept incoming requests to connect and send the login credentials from second tier. The string received in bytes is then deserialized to a login object, which is then verified with the database and according to which different responses are sent back to the client and the socket connection stops.

4.3 Data layer - third tier

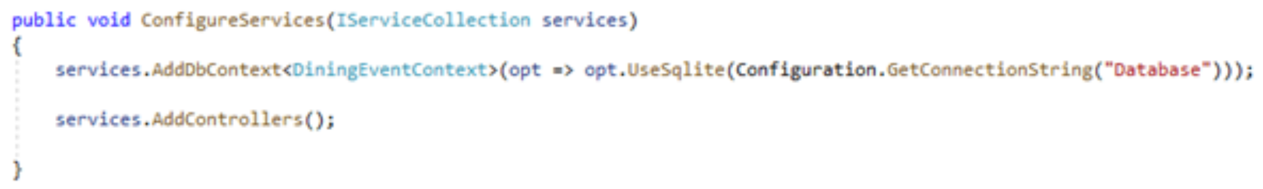
The Database Layer or so-called Data tier consists of a database and a program to read and write information from/to the database. The aim of the data tier is to comprise the database, data storage and data access layer. To ease the communication between the program and the database, Entity framework Core was used.

4.3.1 Database connection

For the database connection SQL Lite database provider has been chosen. SQL lite was used since it does not require any installation and configuration. It has a rather good performance since it reads and writes 35% faster than file system. SQL Lite only reads the parts of the data which are needed directly from the database files previously stored on the disk, rather than reading the entire file and holding it to memory. However, the main reason for choosing SQL Lite was based on the fact that it is a software library that provides relational database management system without requiring a separate server process for operation. In this way “Dining with strangers” does not violate three tier architecture requirements by adding an extra server to the system.

4.3.2 Startup class

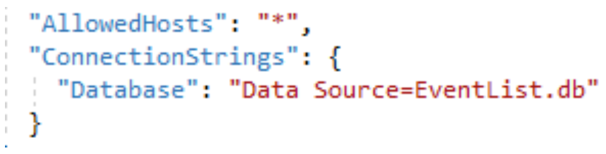
ASP:NET Core apps use Startup class where optionally can be included *ConfigureServices(*) method to configure the app´s services. A service is a reusable component which provides functionality. Services are registered in *ConfigureServices()* and consumed through the application via dependency injection. Additional services were registered with extension methods as it is shown in the picture below. When a service collection method is available to register a service, the protocol is to use a single Add {Service Name} extension method in order to add all required services to the service. The following code shows how to add additional services to the container using extension method *AddDbContext* bond to the *DiningEventContext*. The code allows *DiningEventContext* to use an instance which represents a session with the database using SQL Lite database provider on the specified connection string “Database”. Then, the same service has been added to the controllers.



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<DiningEventContext>(opt => opt.UseSqlite(Configuration.GetConnectionString("Database")));

    services.AddControllers();

}
```

4.3.3 Appsettings.json file

To prevent hardcoding from the system connection string is being assigned to a reference called “Database”. The connection string reference was created in JSON. Initiation of the connection string can be seen below the picture.



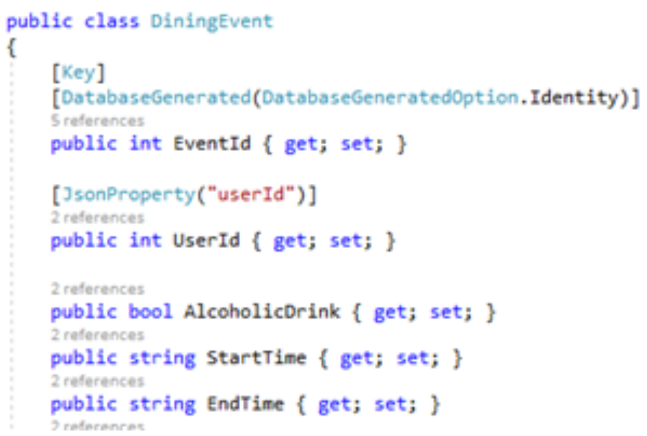
```
"AllowedHosts": "*",
"ConnectionStrings": {
  "Database": "Data Source=EventList.db"
}
```

4.3.4 EF Core

To enable the communication between the database and the data layer, Entity Framework Core has been added to the system. Entity Framework Core is a modern object-mapper for .NET which supports change tracking, updates, schema migrations and LINQ queries. Code first approach was used to build the database tables and relations. Meaning the database were built based on the code which was implemented inside the data layer.

4.3.5 DiningEvent class

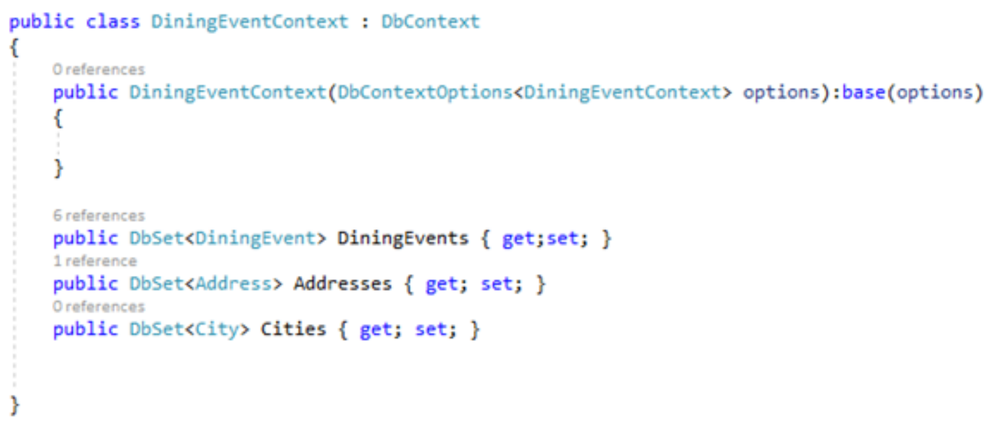
For the EF Core code first approach, entity classes were created which are representing the database tables and attributes with the needed types and constrains. As it is shown below, *DiningEvent* class has multiple attributes which determines the table columns. Notice that *EventId* has the keyword [Key] which means it is going to be the primary key of the *DiningEvent* table inside the database. *EventId* has a second keyword which provides an auto filter functionality to the field attribute.



```
public class DiningEvent
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    5 references
    public int EventId { get; set; }

    [JsonProperty("userId")]
    2 references
    public int UserId { get; set; }

    2 references
    public bool AlcoholicDrink { get; set; }
    2 references
    public string StartTime { get; set; }
    2 references
    public string EndTime { get; set; }
    2 references
```

4.3.6 DiningEventContex class

In EF Core the data access is established using a model that is in the database layer called *DiningEventContext* class. The *DiningEventContext* class is made up of entity classes and a context object what represents a session with the database, permitting the system to use query and save data. Entity Framework uses a set of conventions to build a model based on the shape of the entity classes. As it shown below User, *DiningEvent*, Address and Cities are the entity classes in Dining with strangers. The *DbSet<DiningEvent>* is used to query and instance of the *DiningEvent* object.



```
public class DiningEventContext : DbContext
{
    0 references
    public DiningEventContext(DbContextOptions<DiningEventContext> options):base(options)
    {

    }

    6 references
    public DbSet<DiningEvent> DiningEvents { get;set; }
    1 reference
    public DbSet<Address> Addresses { get; set; }
    0 references
    public DbSet<City> Cities { get; set; }


}
```

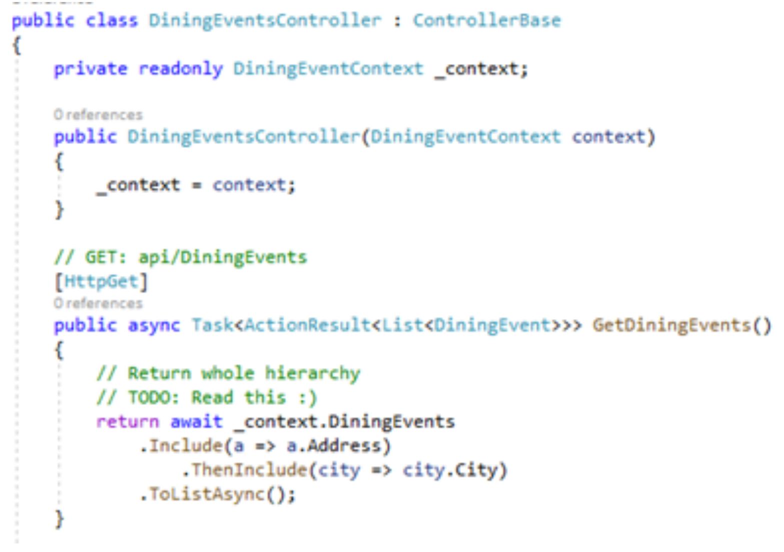
4.3.7 REST API

Rest API was used to enable two separate tiers to communicate with one another. An API defines the correct way to request services or expose data within different contexts and across multiple channels. Restful API is a program interface which uses http requests to GET, PUT, POST or DELETE data.

4.3.8 DiningEventsController class

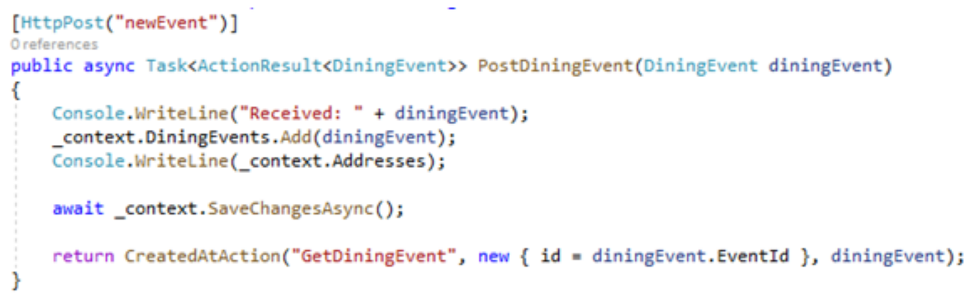
EF Core uses LINQ to query data from the database. LINQ allows the system to use strongly typed query code in C#. Derived context and entity classes are used to reference database object from the *DiningEventContext* class. All *_context* instance has a *ChangeTracker* that is responsible for following changes that need to be written in the database. When a change has been made to instance of the entity classes, the changes are recorded by the *ChangeTracker* and then written to the database when the *SaveChanges()* method is called. Then SQL Lite is responsible for translating the changes into the database-specific operations such as INSERT, UPDATE and DELETE.

When the *getDiningEvents()* called it starts to expose a list of *DiningEvent* objects where the Address and City objects are included. With HTTP GET the corresponding data is accessible through URL in JSON format.



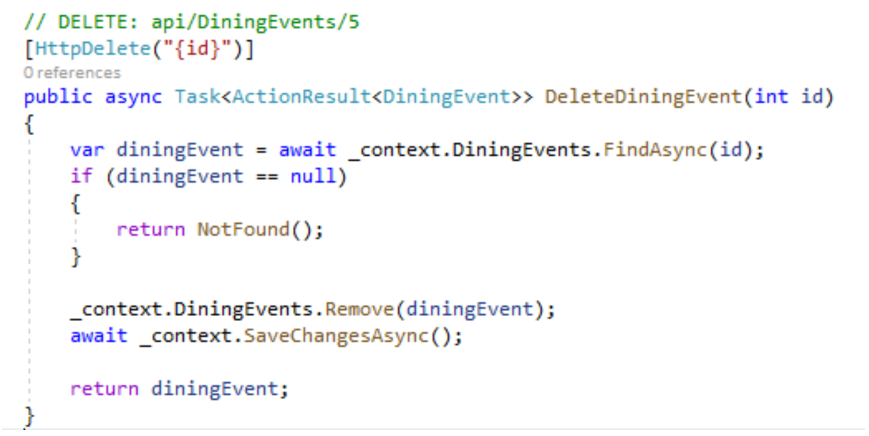
```
public class DiningEventsController : ControllerBase
{
    private readonly DiningEventContext _context;

    0 references
    public DiningEventsController(DiningEventContext context)
    {
        _context = context;
    }

    // GET: api/DiningEvents
    [HttpGet]
    0 references
    public async Task<ActionResult<List<DiningEvent>>> GetDiningEvents()
    {
        // Return whole hierarchy
        // TODO: Read this :)
        return await _context.DiningEvents
            .Include(a => a.Address)
                .ThenInclude(city => city.City)
            .ToListAsync();
    }
```

When an HTTP Post API request is called, the controller consumes JSON to add new information to the database. *PostDiningEvent()* method waits for the needed object which is defined in the argument as a *DiningEvent* object. After the method is executed it writes the received Dining event to the console than add it to the tables. It prints out the address information as well and saves all the changes to the database using *SaveChangesAsync()* method on *_context*. At last, the method returns the view of the newly added dining event.



```
[HttpPost("newEvent")]
0 references
public async Task<ActionResult<DiningEvent>> PostDiningEvent(DiningEvent diningEvent)
{
    Console.WriteLine("Received: " + diningEvent);
    _context.DiningEvents.Add(diningEvent);
    Console.WriteLine(_context.Addresses);

    await _context.SaveChangesAsync();

    return CreatedAtAction("GetDiningEvent", new { id = diningEvent.EventId }, diningEvent);
}
```

HTTP Delete is used to drop columns from the database. As the *DeleteDiningEvent()* method shows below it waits for an int id as an argument in order to find the needed dining event. In order to find it a variable was created where through the *DiningEvent* table the system searches based on the given id. If the id does not exist system returns Not Found() method which produces a status code, status 404 not found response. Otherwise, through *_context* the previously found dining event object will be deleted from the *DiningEvent* table as the remove order has been called. Then, database is updates by calling the *SaveChangesAsync()* on *_context*. The method returns the deleted event object.



```
// DELETE: api/DiningEvents/5
[HttpDelete("{id}")]
0 references
public async Task<ActionResult<DiningEvent>> DeleteDiningEvent(int id)
{
    var diningEvent = await _context.DiningEvents.FindAsync(id);
    if (diningEvent == null)
    {
        return NotFound();
    }

    _context.DiningEvents.Remove(diningEvent);
    await _context.SaveChangesAsync();

    return diningEvent;
}
```

5 Test

Postman was used for testing second and third tier. The java API has been tested by sending data format expected in first tier and check results. The Postman software uses REST HTTP request to test fetching and posting information from APIs. This approach allowed to test and validate the responses (Postman | The Collaboration Platform for API Development, 2019).

The system has been tested using black box testing method (see Appendix H: Test cases). The three tiers have been tested independently throughout the whole project duration after implementing new functionality. Following tests were conducted with the use of all three tiers and were focused on detecting interface errors, accessing database from other layers and fetching data, performance errors, rendering data from the database, faulty or missing functions. The black box testing of the system was based on verification of the system outputs against the expected result. Black box tests with the use of three tiers were performed for the following use cases:

- Create account
- Create event
- View account details
- Search events
- View event details
- View history of events

The following table contains scenarios for testing Create event feature:

Action no.	Action	Reaction
1.	Select create event	Verify that the event creation interface is displayed.
2.	Fill the fields with data	Verify that the system is recording user inputs.
3.	Select Create button	Verify that the system asks for confirmation.
4.a.	Select Create	Verify that the event cannot be created with mandatory fields being empty. Verify that you cannot create an event with a given date, if you already have an event scheduled for that date. Verify that you cannot create an event with the date being in the past. Verify that you cannot create an event with maxNoOfGuests being 0 or negative. Verify that the age limit cannot be negative.
4.b.	Select Cancel	Verify that you're returned to the create event interface.
5.	Await confirmation of event being stored.	Verify that the system confirms the event creation.
6.	Cancel event creation.	Verify that you're returned to the main page.

Test step	Test data	Expected result	Actual result	Notes
User inputs: City Post code Street Block number Floor number Flat number Date Start time End time Number of guests	Horsens 4000 Happy Feet 4 2 1 9/4/2019 12:30 18:30 6 26	No error message is displayed, all the fields have green outline	Passed	

Age limit Pets Description Entry fee Entertainment Drinks Starter Main course Dessert	No Birthday party 100 Yes Yes Soup Pizza Cake			
User presses Create button		Confirmation prompt appears	Passed	
User presses Create button		User is navigated to Search events page, successful message appears	Not passed	Rendering page error
		Database entry appears	Passed	

6 Results and Discussion

The result of the project development was achieved by implementing and testing 4 user stories throughout the three tiers with desired outcome.

- As a Viewer I want the ability to create an account, so I can participate in the community.
- As a viewer I want the ability to see a list of dining events, so that I can see if I'm interested in the community.
- As a user I want the ability to search for events, so I can see a selection of events matching my criteria.
- As a user I want the ability to write a description for my event, so that I can inform other users about the event and its contents.
- As a user I want the ability to create an event, so I have the chance of dining with strangers.
- As a user I want the ability to set a limit on how many users can join my event, so that I can plan accordingly.

7 Conclusions

This semester project was built by analyzing user stories into functional and non-functional requirements, prioritizing the most important functionalities and for more clarification they were translated to use case diagram, use case descriptions, activity diagrams, system sequence diagram and analysis domain model. Analysis is an important part of the foundation of the implemented system as it then let to the design choice of all three tiers. The design is the map of how the application will work based on the analysis, In this phase the class diagrams and sequence were made considering the heterogeneous system architecture. There were a few flaws in the analysis as the implementation had two different models but this turned out to be beneficial to add some logic to the business layer rather than just acting as pipeline to pass objects but also extracting and converting objects to other models. The final product was tested by different scenarios in black box testing and white box testing. To conclude, the application has a four user stories implemented and tested to a satisfactory level.

8 Project future

Project Future Since dining with strangers is not a fully developed product, there is more space for further development to complete the final product to its best version. The most significant user stories to be added to the system is, the ability of a user to send or receive requests to a dining event. Chat system could ease communication between the users. Extra functionalities related to account setting and password would provide more security to the system and its users. Later on the theoretical view of security, documented in the project report can also be considered to be implemented along with other functionalities.

9 Sources of information

Note: Use the standard reference method: Harvard Anglia. A very good reference tool is Mendeley (Mendeley.com 2016), ask VIA Library if you need help.

Blackdoginstitute.org.au. (2018). What is loneliness and how can we overcome it? Explained. [online] Available at: https://www.blackdoginstitute.org.au/news/news-detail/2018/11/12/what-is-loneliness-and-how-can-we-overcome-it-explained [Accessed 27 Sep. 2019].

Visual-paradigm.com. 2019. *Conceptual, Logical And Physical Data Model*. [online] Available at: <https://www.visual-paradigm.com/support/documents/vpuserguide/3563/3564/85378_conceptual,l.html> [Accessed 19 December 2019].

Department for Digital, Culture, Media and Sport (2019). A connected society - A strategy for tackling loneliness. London: Department for Digital, Culture, Media and Sport, p.17. Available at: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/750909/6.4882_DCMS_Loneliness_Strategy_web_Update.pdf [Accessed 27 Sep. 2019]

Hrsa.gov. (2019). The "Loneliness Epidemic" | Official web site of the U.S. Health Resources & Services Administration. [online] Available at: https://www.hrsa.gov/enews/past-issues/2019/january-17/loneliness-epidemic [Accessed 27 Sep. 2019].

Dingle, D. (2019). Tackling the growing problem of loneliness and isolation. [online] UQ News. Available at: https://www.uq.edu.au/news/article/2019/07/tackling-growing-problem-of-loneliness-and-isolation [Accessed 27 Sep. 2019].

Lasgaard, M., Christansen, J. and Friis, K. (2019). Ensomhed blandt unge. [ebook] Region Midtjylland, pp.14, 15, 24, 25, 26. Available at: https://www.maryfonden.dk/files/Ensomhed%20blandt%20unge.pdf [Accessed 27 Sep. 2019].

Ardelt, M. and Ferrari, M. (2018). Effects of wisdom and religiosity on subjective well-being in old age and young adulthood: exploring the pathways through mastery and purpose in life. [online] Cambridge.org. Available at: https://www.cambridge.org/core/journals/international-psychogeriatrics/article/effects-of-wisdom-and-religiosity-on-subjective-wellbeing-in-old-age-and-young-adulthood-exploring-the-pathways-through-mastery-and-purpose-in-life/C59037134A43FD9507E06AFD45580E84 [Accessed 27 Sep. 2019].

T. Cacioppo, J. and Elizabeth Hughes, M., 2006. Loneliness As A Specific Risk Factor For Depressive Symptoms: Cross-Sectional And Longitudinal Analyses. [online] Citeseerx.ist.psu.edu. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.321.3922&rep=rep1&type=pdf> [Accessed 27 September 2019].

Maryfonden.dk. 2016. Hver 4. Dansker Savner Nogen At Spise Sammen Med | Maryfonden.Dk. [online] Available at: <https://www.maryfonden.dk/da/danmark-spiser-sammen-pm> [Accessed 30 September 2019].

Korsgaard, K., 2016. Danmark Spiser Sammen. [PDF] MaryFonden, p.14. Available at: <https://www.maryfonden.dk/files/files/Danmark%20spiser%20sammen_final.pdf> [Accessed 30 September 2019].

Carlton, K. (2019). What is loneliness? - UChicago Medicine. [online] Uchicagomedicine.org. Available at: https://www.uchicagomedicine.org/forefront/health-and-wellness-articles/2019/february/what-is-loneliness [Accessed 27 Sep. 2019].

Sörman DE, e. (2019). Social relationships and risk of dementia: a population-based study. - PubMed - NCBI. [online] Ncbi.nlm.nih.gov. Available at: https://www.ncbi.nlm.nih.gov/pubmed/25779679 [Accessed 29 Sep. 2019].

R, M. (2010). The therapeutic value of laughter in medicine. - PubMed - NCBI. [online] Ncbi.nlm.nih.gov. Available at: https://www.ncbi.nlm.nih.gov/pubmed/21280463 [Accessed 29 Sep. 2019].

Umberson, D. and Karas Montez, J. (2019). Social Relationships and Health: A Flashpoint for Health Policy.

Banger, D., 2014. A Basic Non-Functional Requirements Checklist « Thoughts from the Systems front line.... Available at: https://dalbanger.wordpress.com/2014/01/08/a-basic-non-functional-requirements-checklist/ [Accessed January 31, 2017].

Business Analyst Learnings, 2013. MoSCoW : Requirements Prioritization Technique — Business Analyst Learnings. , pp.1–5. Available at: https://businessanalystlearnings.com/ba-techniques/2013/3/5/moscow-technique-requirements-prioritization [Accessed January 31, 2017].

Dawson, C.W., 2009. Projects in Computing and Information Systems, Available at: http://www.sentimentaltoday.net/National_Academy_Press/0321263553.Addison.Wesley.Publishing.Company.Projects.in.Computing.and.Information.Systems.A.Students.Guide.Jun.2005.pdf.

Gamma, E. et al., 2002. Design Patterns – Elements of Reusable Object-Oriented Software, Available at: http://books.google.com/books?id=JPOaP7cyk6wC&pg=PA78&dq=intitle:Design+Patterns+Elements+of+Reusable+Object+Oriented+Software&hl=&cd=3&source=gbs_api%5Cnpapers2://publication/uuid/944613AA-7124-44A4-B86F-C7B2123344F3.

IEEE Computer Society, 2008. IEEE Std 829-2008, IEEE Standard for Software and System Test Documentation,

Larman, C., 2004. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,

Mendeley.com, 2016. Homepage | Mendeley. Available at: https://www.mendeley.com/ [Accessed February 2, 2017].

YourCoach, S.M.A.R.T. goal setting | SMART | Coaching tools | YourCoach Gent. Available at: http://www.yourcoach.be/en/coaching-tools/smart-goal-setting.php [Accessed August 19, 2017].

Docs.microsoft.com. 2019. *Idisposable Interface (System)*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netframework-4.8> [Accessed 17 December 2019].

Docs.microsoft.com. 2019. *Dependency Injection In ASP.NET Core*. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1> [Accessed 15 December 2019].

Walker, S., 2019. Oqtane Blog | *Exploring Authentication In Blazor*. [online] Oqtane.org. Available at: <https://www.oqtane.org/Resources/Blog/PostId/527/exploring-authentication-in-blazor?fbclid=IwAR0rbQkY47cHHxs29HWCk0RggH7GHeLDx3kJ4vwmgUGMTsFU3hxpsQ9ybZo> [Accessed 19 December 2019].

SearchSoftwareQuality. (2019). *What is a 3-Tier Application Architecture?* - Definition from WhatIs.com. [online] Available at: https://searchsoftwarequality.techtarget.com/definition/3-tier-application [Accessed 18 Dec. 2019].

SearchAppArchitecture. (2019). *What is a RESTful API (REST API) and How Does it Work?*. [online] Available at: https://searchapparchitecture.techtarget.com/definition/RESTful-API [Accessed 18 Dec. 2019].

SQLite Tutorial. (2019). *What is SQLite? Top SQLite Features You Should Know*. [online] Available at: https://www.sqlitetutorial.net/what-is-sqlite/ [Accessed 18 Dec. 2019].

Postman. 2019. Postman | The Collaboration Platform For API Development. [online] Available at: <https://www.getpostman.com> [Accessed 20 December 2019].