

University College of Northern Denmark

Technology and Business

Computer Science Academy Profession (AP) Degree

Class: DMAJ0916

Title: Technology and Programming Report – 3rd Semester 2017

Project participants (Group 2):

-Ralfs Zangis

-Andrei-Eugen Birta

-Hannes Heiskonen

-Stoycho Nenov Anastasov

Supervisors:

-Brian Hvarregaard

-Per Trosborg

Due date: 2017-December-18 (13:45 UTC+1)

Submission date: xxxx

Birta Zanigis Heiskonen Nenov

[1. Preliminary Study 3](#_Toc500503569)

[1.1. Introduction 3](#_Toc500503570)

[1.2. Project Idea 3](#_Toc500503571)

[1.3. Problem Statement 3](#_Toc500503572)

[2. Development Process 3](#_Toc500503573)

[2.1. Service Related 3](#_Toc500503574)

[2.1.1. Type of Service 3](#_Toc500503575)

[2.1.2. Framework 4](#_Toc500503576)

[2.1.3. Service’s Architecture 5](#_Toc500503577)

[2.1.4. Database Architecture 6](#_Toc500503578)

[2.2. Client Related 8](#_Toc500503579)

[2.2.1. Dedicated Client 8](#_Toc500503580)

[2.2.2. Web Client 8](#_Toc500503581)

[2.3. Middleware 9](#_Toc500503582)

[2.3.1. Client-Service 9](#_Toc500503583)

[a. Choice 9](#_Toc500503584)

[b. Pros 9](#_Toc500503585)

[c. Cons 9](#_Toc500503586)

[d. Other choices 9](#_Toc500503587)

[2.3.2. Service-Database 9](#_Toc500503588)

[2.4. Security 9](#_Toc500503589)

[2.4.1 Password storage 9](#_Toc500503590)

[2.4.2 SQL Injection 10](#_Toc500503591)

[2.5. Concurrency 11](#_Toc500503592)

[2.6. Performance 11](#_Toc500503593)

[2.7. Interesting bits of code 12](#_Toc500503594)

[a. Keeping connection from timing out 12](#_Toc500503595)

[b. Callback methods in chat 12](#_Toc500503596)

[c. Database Triggers 12](#_Toc500503597)

[3. Conclusion 12](#_Toc500503598)

[3.1. Denouement 12](#_Toc500503599)

[3.2. References 13](#_Toc500503600)

# Preliminary Study

## Introduction

This document summarizes the collaboration of Group 2 for the programming and technology exam of the 3rd Semester. The group consists of 4 members of 4 different nationalities. Despite the major differences in our opinions (we agreed on certain rules and guidelines to follow, thoroughly elaborated in the accompanying document called group contract), we managed to harness the benefits of diverse ideas and identify multiple possible approaches to certain problems.

## Project Idea

The idea of the project is to create a service that handles multiple chatrooms with limited people count capacity, where people can discuss topics they are passionate about; which the users may access either via the web client or the windows application. Listening to music through the application using YouTube Data API, joining as groups of people or simply playing a game of rock-paper-scissors, are other major user stories of our program.

## Problem Statement

Some of the problems we had to find the answer to, include but are not limited to: Users whose number is greater than the available slots in a given chatroom try to join at the same time (for example, there is 1 spot left but 2 people try to join), a group whose number of people is greater than the available slots in a given chatroom tries to join (for example, a group of 4 attempts to join when there are only 3 or less spots left, preventing all of them from joining), creating a safe, secure and stable program, learning how to use callbacks and bindings.

After formulating the problem statement and having it approved by the supervisors we were assigned the task of solving the problem by creating a web service, selecting the appropriate middleware and create both a dedicated and a web client.

# Development Process

Considering the fact that the main idea of this project was to design and implement a web service, choosing the right framework to follow is one of the most important parts of it.

## Service Related

### Type of Service

When we took the decision on which API to follow, two main choices were on the table: Representational State Transfer (REST) and Simple Object Access Protocol (SOAP).

REST is not bound to a single protocol, which allows for greater extensibility of our software. Also, REST has relatively low degree of coupling between the client and the service allowing better maintainability for our service. Lastly, REST is stateless – messages exchanged between the server and the client have all the necessary information for the message to be processed. However, having all this extra data in the messages can be considered redundant information (in the case where that some or all of this information is not used) which can have a small negative impact on latency.

SOAP which is not architecture but a protocol (as it can be seen from the name), on the other hand, if well implemented could offer slightly better performance. However, the knowledge and experience required to use it properly would be increasing both development and maintenance costs and decrease scalability, because of the growing coupling.

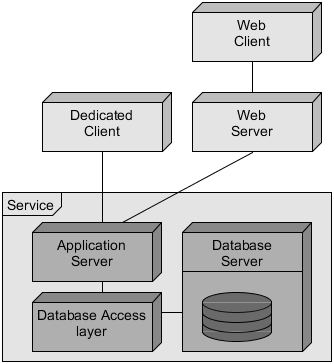
All of the above-mentioned reasons, determined us to create a RESTful service.

### Framework

In terms of framework, we were split between WCF and WebAPI. But since our goal was first heterogeneity and second speed, we chose WCF, simply because it can have multiple types of bindings including TCP which can be faster, has binary data format, lower compatibility level; and HTTP, which has xml data format, has greater compatibility level. Furthermore, it offers great flexibility as it automatically selects the appropriate type of binding depending on what the user device is compatible with (as long as such type of a binding is configured).

The main reasons we even considered WebAPI was because it is easier to understand and learn, it is great for HTTP services, and because it can be faster than WCF. However the difference in speed, at least for the number of users we anticipate, is so insignificant that its not worth the trade off with the compatibility WCF offers.

### Service’s Architecture

Our service’s architecture is a multi-tier architecture (as can be seen in Fig.1), which provides several benefits, such as easy expandability (for example adding a mobile client) and low-cost maintainability. Not only that, but also, this architecture helps us achieve the goal we have set for ourselves, for this project, that being to pursue “high cohesion and low coupling”. Other possible architectures which we could’ve decided to choose were the classic web architecture and client/server architecture, both being dismissed because of their lower level of flexibility.

Our domain model (Fig.2) was one of the few diagrams we have create for the project. Much thought and time has been invested in its creation; and the following bullet points should answer any questions related to it.

* Profile- all user related information.

Figure

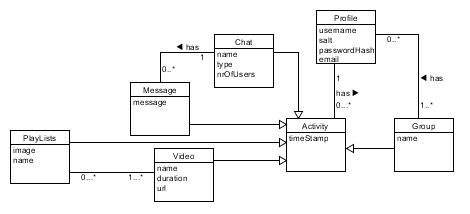
* Activity- holds a timestamp and reference to user, to know which user, what and when, they created something.
* Group- depicts a premade team/fellowship of users that want to do things together.
* Chat- holds the location (chatroom) where messages will be displayed.
* Message- holds the actual text written by a user.

Figure 2

* Video- holds the reference of video to be played.
* Playlist- holds user created playlists.

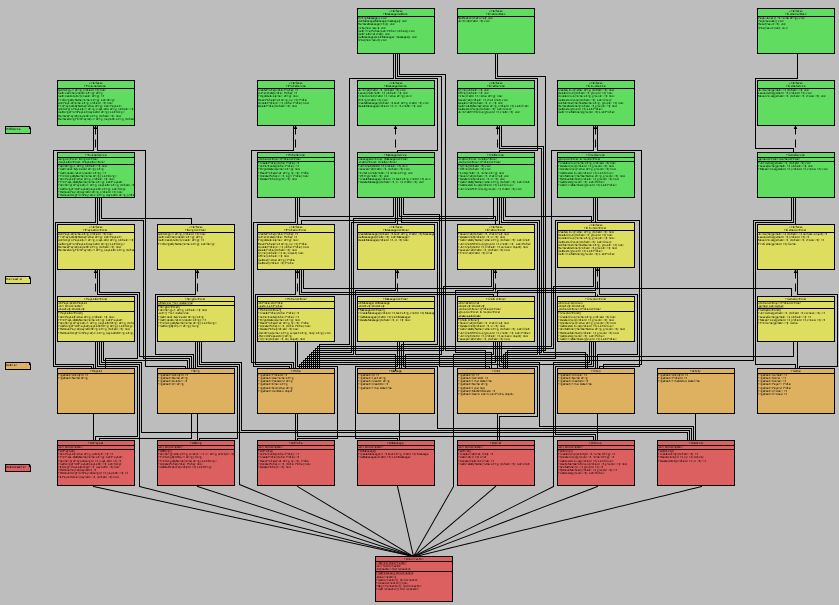
As you can see in our design class diagram, for the service (Fig.3 although hard to read, we invite you to see it in our included files “DesignClassDiagram.uxf”), we have decided to organize the code in several “tiers” (depicted by a different color, in the figure), to help us identify where a specific class should be.

Figure 3

Also in order to improve maintainability and ease future changes, we added interfaces to both controller and service classes.

### Database Architecture

A database is an organized collection of data. Databases can be designed to do just about anything with information – Track, organize, edit and produce reports. Many great applications and websites have a database as a main structural part.

There are two types of database technologies: Relational Databases, which are great at organizing and retrieving structured data; and Non-Relational Databases, which are best used when the data is inconsistent, incomplete or simply massive.

For our project, we have chosen to go with a Relational Database, simply because pros such as: strict ACID support, data normalization, supports joins, limitless indexing, and being one of the most common used technologies\*, outweigh the cons of having a non-relational database, cons such as: working with joins can be difficult, slow mass updates, difficulty tracking schema changes.

As for engines, there are several choices that we considered, for a Relational Database, some of which are: Oracle Database, SQL Server and MySQL; and since all three of them were using dialects of the same language (SQL), it went down to the very basics when we took the decision on which to use.

As a final decision, we chose SQL Server 2014, because of the following: SQL Server executes and commits each instruction, unlike Oracle which requires explicit command to commit the changes; ease of use, since not only were we thought on how to use it, but also compared to Oracle, which give so many other settings and configurations that can be set to the wrong value; and performance.

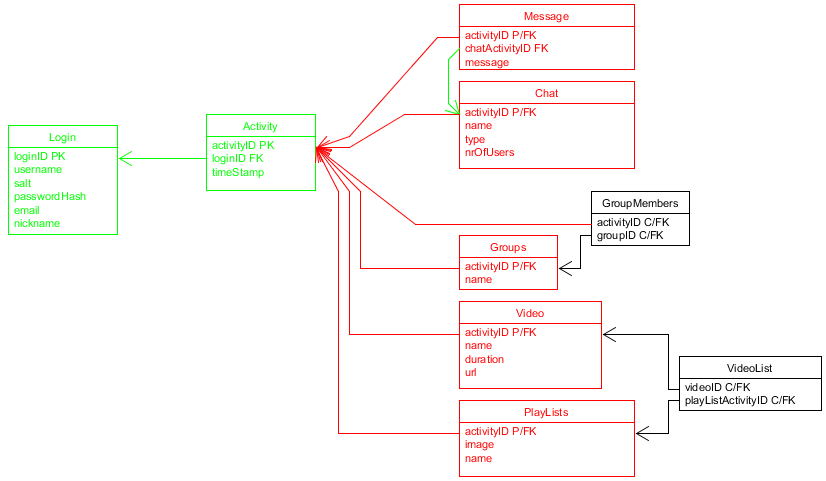


Figure 4

As you can see in Fig.4, Database Diagram, we have 3 types of tables: tables which hold critical information such as user details and actions (green), tables which specify the type of activity a user has done (red) and tables which’s main purpose is to handle a M-M relationship (black).

## Client Related

In order to have a successful web service we needed something to consume the service, that being clients. For this project we have decided to create two types of clients: a dedicated windows compatible client and a multi-platform web client.

### 2.2.1. Dedicated Client

For this type of client we had to choose between several options, some of which were: WinForms, wpf and xaml.

Our choice was WinForms. We chose to go with it for many reasons. First, for us, design is not so important as functionality. Second, we did not want to spend time on a spike about new ways to create dedicated client. Third, it was made pretty clear that majority of the points which count towards project’s grade come from the backend of the application. Finally, we were already familiar with a java version of WinForms, in eclipse, called “Window Builder”.

Following a popular trend in programming, we decided to implement all of the features in the dedicated client first, then, after finding errors and exceptions and fixing them, to “correctly” implement the features in the web client.

### 2.2.2. Web Client

Just as for the dedicated client, we had to choose between several options, as well. Some of our options were: mvc, web forms and web pages.

We have chosen ASP.NET MVC. At first, we eliminated Single page Application and Web pages, because it was that our application needs definitely more than 1 web page. Arguments against Web Pages were that we have never worked with web pages before and we did not want to do spike on it either. We decided to go with Web Forms, at first. It was easy to implement, user-friendly and works similarly to WinForms, which we used to create desktop application earlier. But after electing a new product owner, the decision to switch to MVC was made, simply because in the process of creating the web client, we became intrigued on the twists we have encountered so far. Also, MVC being more frequently used in actual businesses, than web forms only gave us more reasons to make the switch.

## 2.3. Middleware

### 2.3.1. Client-Service

#### a. Choice

#### b. Pros

#### c. Cons

#### d. Other choices

### 2.3.2. Service-Database

In order to save data in our database of choice (SQL Server 2014), we needed something to help C# communicate with it, and for that we had taken into consideration two possibilities: ADO.NET and ADO.NET Entity Framework.

On one hand ADO.NET Entity Framework has better security, data encapsulation and helps reducing the redundant code, all of which were strong reasons to consider using it, however cons such as slightly slower performance and lack of optimization in terms of a flexible database model, determined us to go with ADO.NET, which even tough has bigger queries and is slightly harder to use, does not require to know the database model beforehand, making it perfect for an agile method of development.

## 2.4. Security

### 2.4.1 Password storage

This is the first step we took towards security measures in our project. The last thing we want is if someone manages to compromise the database to be able to obtain the passwords of the unsuspecting users. What we did was to add ‘salt’ (a unique random string) to the password and then hash it before storing it in the database. This ensures that even if someone manages to see all the values in the database they will not be able to make much use of it as the password is hashed. And because of the salt two same passwords will still have different hash values making it even harder to crack. Of course, we have to store the salt in the database because it is added to the password every time before it is hashed and checked against the password hash stored.

### 2.4.2 SQL Injection\*\*\*

SQL Injection is a wide-spread way of executing malicious code on a database. It occurs when a user enters data in such a way that it executes SQL commands that are not supposed to be executed.

Figure 5

For example, as you can see in Fig. 5, any user introducing such type of statements could potentially drop the database, or even worse, get someone else’s private information, essentially transforming our service into malware.

Declaring an SQL command that takes only parameters (as can be seen in Fig.6) rather than strings, prevents users with bad intentions of executing any code that they should not. Not only is this safer, but it also makes the program more robust by returning an integer of how many rows were affected by this statement. In this way we can easily check if the statement was successfully executed. And if that weren’t enough, it makes the code way more readable and easier to write instead of having a large number of concatenated strings.

Figure 6

## 2.5. Concurrency

In this project we have three places where our users will have to “fight” for a place:

* Join chat (as a single user)
* Join chat with group (either as part of a group or as the leader of the group)
* Join game

We decided to approach this problem using the pessimistic way of handling it. Why the pessimistic approach? Because we found it being the best way of preventing the lost updates problem.

The concurrency between users would happen like this: two users try to join chat where max number of places is 5, 4 of which being occupied. Both, user1 and user2 would see the chatroom having one available place and both would try to join it at the exact same time. Without handling this problem, both users would successfully join the chatroom, making a total of 6 users inside, even though the limit was set to 5. A similar type of issues appears when users would try to join a chatroom with a group, the only difference being that the complexity of the problem would slightly increase considering the fact that either all users in the group have to successfully join or none should be able to.

The way we handle this issue is by using “locks”(Fig.7 shows an example of how we are using locks): after a user starts joining a chatroom, the service locks that specific chatroom until the action is successfully finished. As for joining a chatroom with a group, the same method is used, the only difference being that the method is called for each group member, inside a transaction, and in case one of the users fails to join, the transaction would be rolled back, resulting in no group members being added to the chatroom.

Figure 7

As for the ACID properties of our project, we tried ensuring them as follows:

* Atomicity: by using transactions;
* Consistency: by testing the data before saving it into the database;
* Isolation: by using Repeatable Reads and locks;
* Durability: by catching any exception and giving inputs as parameters.

## 2.6. Performance

Here we have measured the time it takes for certain methods to be executed completely. The tests were conducted using the “System.Diagnostics.Stopwatch” class by starting the timer at the beginning of a button click and printing the result as soon as the action was completed.

The following chart shows the time it took to login with up to 16 online users by sending a login request every second:

Figure 8

Average: 367ms, Best: 335ms, Worst: 407ms.

As you can see the results are very close to each other and the differences can be accounted to outside factors such as current load of the network it goes through, background OS process etc.

Compared to the time it took to join a chatroom as a group of 20 users simultaneously which was 454 milliseconds, we can conclude that there is barely any decay in performance, and safely assume that the service will be able to handle at least 50 concurrent clients.

## 2.7. Interesting bits of code

### a. Keeping connection from timing out

### b. Callback methods in chat

### c. Database Triggers

Since one of the features, that would ease our programming task, we have decided in the begging was cascade deletion, and SQL Server 2014 does not allow such thing when a table is referenced, or references, by multiple foreign keys, we needed to find another way of dealing with phantom data. Shortly after the problem was encountered, we stumbled upon this ingenious way of doing it: Deletion Triggers.

What is a deletion trigger? Well, when a row is deleted from a table, instead of doing the default command, the server would execute this special piece of code, which specifies from which tables to delete and what.



Figure 9

As you can see (Fig.9), before deleting the entry that was commanded to be deleted, the engine would check in all of the mentioned tables if the entry’s foreign key, is referenced, then proceed to delete that entry, before continuing the check, and finally executing the commanded entry’s deletion.

# 3. Conclusion

## 3.1. Denouement

In conclusion, during this semester we have managed to achieve not only knowledge about different frameworks of developing software, a new programming language but also a new development concept, that being programming a distributed service.

As an ending note, we would like to thank all the readers, who invested their time in reading this paper and also to the guiding teachers that helped and guided us through the entire process.

## 3.2. References

\*According to James Serra, Big Data/Data Warehouse Evangelist at Microsoft in a presentation Published on Mar 15, 2016

\*\*According to Microsoft public documentation on Windows-forms and MVC, available at the following links:

* https://docs.microsoft.com/en-us/dotnet/framework/winforms/windows-forms-overview
* <https://docs.microsoft.com/en-us/visualstudio/designers/introduction-to-wpf>

\*\*\*Inspired by an article from W3School on SQL Injections, available at the following link:

* https://www.w3schools.com/sql/sql\_injection.asp

\*\*\*\*Inspired by Microsoft public documentation on isolation levels, available at the following link: <https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels>

<https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/configuring-system-provided-bindings>