Project 1
TDT 4145

Sara Elise Wøllo, Thomas Torkildsen, Markus Trætli

March 2021

1 How to read this project-report

We have chosen to split the project report into two parts, just like the project was presented to us. The first Part is DB1, the first part of the project. The second part om the report is DB2, the second part of the project delivery.

2 Part 1 - ER-model and implementing the database using SQL.

2.1 Introduction

In this project, we are creating a Piazza-forum. This forum has the functionality that we are used to from our studies. Where we have made assumptions about the functionality of the forum, we have specified these. To build our forum, we have first made an ER-model, we have then presented a relational database for our Piazza forum which we have implemented using SQL. Lastly, we have looked at a number of different use-cases of our Piazza-forum, ans shown how these problems are solved in the database. Later, we will implement the Piazza-forum.

2.2 Entity-Relationship model

2.3 The ER-model

2.3.1 Assumptions

- A user can be an instructor in one course and a student in another. For instance, the teaching assistants are instructors in the courses they are TAs for, and students in their regular courses.
- A instructor can invite all users to all courses where he is instructor. *Type* defines the type of invitation (student or instructor).
- There might exist users that are not students nor instructors, for example Admin-people.
- Once a course is created the anonymity setting (*Anon*) can not be changed.
- Each course can have from zero to several folders, but a folder can only belong to one course.
- Each folder can have from zero to several subfolders, but each subfolder can only belong to one folder.
- Each folder can have from zero to several threads, but a thread can only belong to one folder.
- Each thread can have one to several posts, but a post can only belong to one thread.
- Each post can only have one tag.
- Every post can be liked by several users and a user can like several posts. However, a user can only like one post one time, but this is something that is handeled in our database system.
- Every user can post several posts, but a post can only be posted by one user.
- Every folder must be managed once (every folder must have been created) or several times, but not every instructor must manage a folder.
- Once a thread is viewed every post in the thread is automatically viewed as well. This is because once you click on a thread in Piazza, all the posts in the thread are automatically shown on screen.
- A post can link to arbitrarily many posts and the other way around.

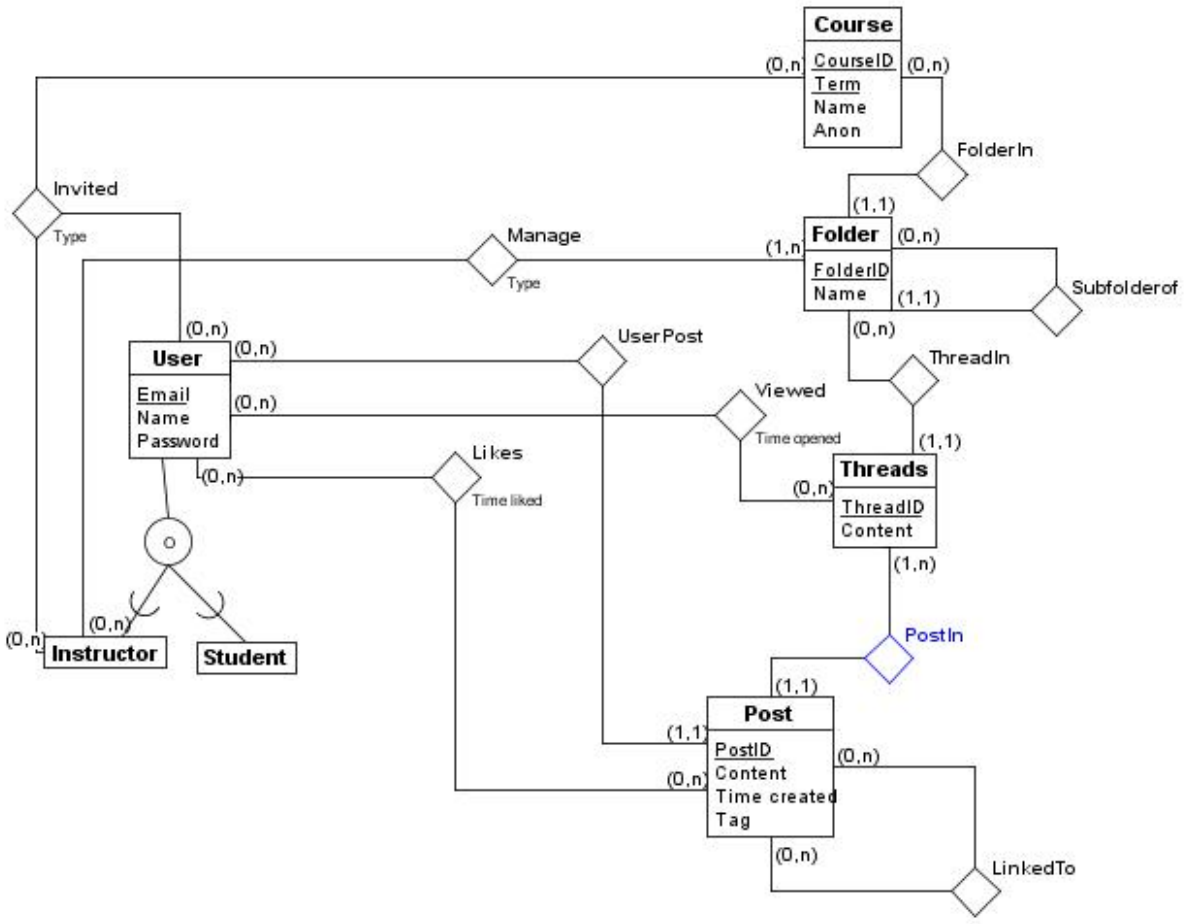

Figure 1: Entity-Relationship model for our Piazza-database

2.4 Mapping the ER-model to a relational database

When chosing the different tables we need in our database, we have focused on all our tables fulfilling the forth Normal Form (4NF).

We have ended up with a database model with 15 tables. The names of these tables are defined in table 1.

Table name
Course
Folder
Thread
Post
Likes
Viewed
Manage
EnrolledIn
Subfolder
UserPost
LinkedTo
UserProfile
FolderInCourse
ThreadInFolder
PostInThread

Table 1: Tables in our relational database

We have defined the different tables below, where the primary keys (or combinations of primary keys) are underlined in the table with a continuous line, and the foreign keys are underlined with a dashed line.

If an attribute is both a primary and a foreign key, it is underlined with both a continuous and dashed line.

The first table we define is called **Course**. This is shown in table 2

CourseID	Term	CourseName	Anon

Table 2: **Course**

Here, the primary key is (*CourseID, Term*). The reason why we need both of these to be the primary key, is because a course has the same ID from year to year, but the lecturer may decide to create a new Piazza forum each year. These two together define a unique primary key. *Anon* denotes whether the instructor has made it possible for the participants in the course to post anonymously in the forum. We have assumed this to be a setting that the instructor can switch on when they make the forum, and that it is not possible to change this setting once the Piazza forum is created. We assume that *CourseName* and *Anon* are independent of each other and functionaly dependent on (*CourseID,Term*), thus we have 4NF.

We then define the table **Folder** in table 3,

FolderID	Name

Table 3: **Folder**

The *FolderID* is the primary key for this table and *Name* is functionally dependent on *FolderID*.

For the **Thread**-table, shown in table 4,

ThreadID	Title

Table 4: **Thread**

we have a primary key consisting of *ThreadID*. *Title* is functionally dependent on *ThreadID*. It is trivial that both table 3 and table 4 is on 4NF.

PostID	Content	PostCreatedAt	Tag

Table 5: **Post**

For the **Post**-table, we have a primary key consisting of one attribute *PostID*. You can only have one tag connected to each post, saved in *Tag*. This attribute can also have a NULL-value. *Content*, *PostCreatedAt* and *Tag* are all functionally dependent on PostID and independent of each other, thus we have 4NF in this table as well.

We need to specify the users of the Piazza forum, and information about them are found in **UserProfile**, in table 6.

Email	UserName	Password

Table 6: **UserProfile**

Here, *Email* is the primary key and *UserName* and *Password* are functionally dependent on *Email*. Since several users can have the same password, but different username and the other way around we have functional independence between *UserName* and *Password*. Thus we have 4NF.

The next table in our database is **Likes**, seen in table 7.

PostID	Email	TimeLiked

Table 7: **Likes**

This table keeps track of instances where a person "Likes" a post, which post they "Liked", and when they "Liked" the post. Here, (*PostID, Email*) all together is the primary key of the table. *Email* is a foreign key referencing the **UserProfile**-table, and *PostID* is a foreign key to the **Post**-table. *TimeLiked* is functionally dependent on *PostID* and *Email*

We then have a table called **Viewed.** This table keeps track of each instance where a user views a thread (and by opening the thread views all the posts in the thread), and logs this. This is used to generate statistics about user and post activity. The table is shown in table 8.

ThreadID	Email	TimeOpened

Table 8: **Viewed**

In the **Viewed**-table, all the attributes together make up the primary key, so that is (*ThreadID, Email, TimeOpened*). This is because one user can open a thread several times, and to register all these instances we include the *TimeOpened* in the primary key. The foreign keys found in the table are *Email*, referencing **UserProfile**, and *ThreadID* referencing the **Thread**-table. There are no functional dependencies in this table.

The next table is **Manage**. This table keeps track of all users that makes changes to folders, and which folder they change. Changes can include making a new folder, deleting a folder or editing the folder. **Manage** is seen in table 9.

Email	FolderID	ActionType

Table 9: **Manage**

In the **Manage**-table, the primary key is (*Email, FolderID, ActionType*). *Email* is a foreign key referencing the **UserProfile**-table, and *FolderID* is a foreign key referencing the **Folder**-table. There are no functional dependencies in this table.

UserEmail	CourseID	Term	InstructorEmail	AsInstructor

Table 10: **EnrolledIn**

For the **EnrolledIn**-table, We use (*UserEmail,CourseID, Term*) as the primary key. Both *UserEmail* and *InstructorEmail* are a foreign keys referencing the **UserProfile**-table. (*CourseID, Term*) is a foreign key referencing the **Course**-table. Here we assume that *InstructorEmail* is functionally dependent on the primary key, ontherwise we would loose 4NF. This means that each user can only be invited to a course once each term. *AsInstructor* is also functionally dependent on the primary key. If a user is invited to a course as student, the user can not be re-invited as instructor.

For the **FolderInCourse**-table shown in table 11,

FolderID	CourseID	Term

Table 11: **FolderInCourse**

The primary key is *FolderID*. *CourseID* and *Term* is functionally dependent on the primary key. This table has two foreign keys. *FolderID* is a foreign key referencing the **Folder**-table, and (*CourseID, Term*) is a foreign key referencing the **Course**-table. We have 4NF because *CourseID* and *Term* can be assumed independent.

For the **SubFolder**, in table 12, we have a primary key consisting of both attributes in the table, (*FolderID, SubFolderID*). This is because we might have sub-subfolders such that the *SubFolder*-attribute fail to be unique. This table signalizes that one folder is a subfolder of another folder.

FolderID	SubFolderID

Table 12: **Subfolder**

Both *FolderID* and *SubFolderID* are foreign keys referencing the attribute *FolderID* in the **Folder**-table.

PostID	Email

Table 13: **UserPost**

For the **UserPost**-table seen in table 13, the primary key is *PostID*, and there are two foreign keys in the table. These are *PostID* which is referencing **Post**, and *Email* referencing **UserProfile**. *Email* is functionally dependent on *PostID*.

The next table in our relational database is **LinkedTo**, and it keeps track of posts that link to other posts. This is so that one for instance can link to a post that answered a question a user asked in a different post.

PostID	ToPostID

Table 14: **LinkedTo**

The primary key for this table is then both the attributes in the table, namely (*PostID, ToPostID*). The table has two foreign keys, each linking to the **Post**-folder. These are *PostID* and *ToPostID*, and they both link to the *PostID*- attribute in the **Post**-table.

To keep track of the relationship between folders and threads, we need **ThreadInFolder**, shown in table 15. This denotes which folder each thread is located, and used *ThreadID* as the primary key. Both these attributes are also foreign keys, *FolderID* references **Folder**, and *ThreadID* references **Thread**. *FolderID* is functionally dependent on *ThreadID*.

FolderID	ThreadID

Table 15: **ThreadInFolder**

Lastly, we need to connect which post is connected with which thread. This is done in table 16, **ThreadInPost**. In this table, *PostID* is the primary key, and *PostID* is a foreign key to **Post**, whereas *ThreadID* is a foreign key to **Thread**. *ThreadID* is functionally dependent on *PostID*.

PostID	ThreadID

Table 16: **PostInThread**

We can conclude that all the tables are on the fourth normal form, with the assumptions made.

2.5 Use-cases for the database

2.5.1 Use case 1

In order for a user to log in, they must enter an email and a password. This information can be obtained from the user in the java-program. The login details can then be used to make a query to the **UserProfile** table and check for any matches. One such query is shown in figure 2. One is considered logged in if the query returns a match, but if it returns an empty table then the login attempt failed. If the query succeeds one will be prompted to select which courses piazza one would like to see.

```
select * from UserProfile where Email="person@mail.no" and UserPassword="passord123";
```

Figure 2: Example query for logging in

2.5.2 Use case 2

An instructor has made the the folder "Exam" with FolderID=1 belonging to some course. Now, for a student to make a new post in this folder, they have to be logged in and enrolled in this course. Then one has to obtain the title, content and tag from user input. Since this is a post posing a question, one will first have to make a thread and then make a post attached to that thread. We also add the time it was created and add the relation to a different table to keep track of which users wrote which posts across different forums. Using the provided information, one can insert the new post using the code shown in figure 3.

2.6 Use case 3

In order to make a reply one has to make a post in the same thread as the post one wants to reply to. Thus when making a reply, all information about folder, course, thread etc is known. Adding a reply to a post will then simply just adding another post in the same thread.

```
insert into Thread values (1,"Title"); #ThreadID, Title
insert into ThreadInFolder values (1,1); #ThreadID, FolderID. FolderID = 1 is a folder named Exam
insert into Post values(1,"Hey...","2021-11-03-10:00:00","Question");#PostID, Content, TimeDate, Tag
insert into PostInThread values(1,1); #PostID, ThreadID
insert into UserPost values(1,"person@mail.com"); #PostID, Email
```

Figure 3: Insert statements that inserts a new question.

```
insert into Post values(2,"My reply is...","2021-11-03-11:00:00",null); #PostID, Content, TimeDate, Tag
insert into PostInThread values (2,1); #PostID, ThreadID
insert into UserPost values(2,"instructor@mail.com"); #PostID, Email
```

Figure 4: Insert statements that inserts a reply.

2.7 Use case 4

In order to search for posts that contains a certain keyword, one can make a query like in figure 5. One may also add conditions for a specific *Term* and *CourseID* if one is looking in a specific piazza forum.

```
select PostdID from Post
natural join PostInThread
natural join ThreadInFolder
natural join Folder
natural join FolderInCourse
natural join Course
where Content like "%WAL%" and CourseID = "TDT-4145" and Term = "V21";
```

Figure 5: Query that finds the PostID for all posts in the spring 2021 TDT-4145 piazza that contains "WAL"

2.8 Use case 5

An instructor wants to find statistics for users and how many posts they have created and read in a certain course, and sort this my number of posts viewed. The database allows for this information to be found by using a a series of subqueries. Firstly, we find information about all Piazza users and the courses they are enrolled in in the **EnrolledIn**-table. In our first subquery, we use natural join to join together the **Viewed, ThreadInFolder** and **FolderIn**, and aggregate the resulting table on *CourseID, ThreadID, Term, Email* to find the number of views each user has on posts in each course they are taking. We do a similar subquery to find the number of posts each user enrolled in a course has posted in the course. We left join the two subqueries to our **EnrolledIn**-table, so that we have information on all users enrolled in all courses, the number (if any) posts they have created in each course, and the number of threads they have viewed (if any). To get the statistics for one specific course, we use a where-clause on our SQL-query, and finally we order by the number of viewed threads.

```
select tab1.Email, Nviewed, Nposted
from (select Ei.UserEmail as Email, EI.CourseID, EI.Term, NViewed from EnrolledIn as EI
left join
    (select temp.Email,temp.CourseID ,temp.Term, temp.ThreadID, count(ThreadID) as Nviewed
    from
        (select Email, CourseID, Term, ThreadID
            from FolderInCourse
            natural join ThreadInFolder
            natural join Viewed)
        as temp group by temp.CourseID,temp.Term, temp.ThreadID, temp.email) as tab1
    on EI.UserEmail = tab1.Email and EI.CourseID = tab1.CourseID and EI.Term = tab1.Term) as tab1
left join
(select temp.Email,temp.CourseID,temp.Term,count(PostID) as Nposted
from (select Email, CourseID, Term, PostID
    from FolderInCourse natural join ThreadInFolder
    natural join Thread natural join PostInThread
    natural join Post natural join UserPost) as temp
    group by temp.CourseID, temp.email, temp.term) as tab2
on tab1.CourseID = tab2.CourseID and tab1.Email=tab2.Email
where tab1.CourseID = "TDT-4145" and tab1.Term = "H21"
order by Nviewed desc;
```

Figure 6: Query that finds the number of posts created and viewed in a given course.

2.9 Discussion and Conclusion

In our database-model, we emphasised getting tables on the forth normal form. This is useful, because we are then less likely to encounter problems with redundancy.

In a large database system however, we will need to break down our information into many small tables to keep the forth normal form. This makes the process of compiling information from the database more tedious, as we will need to join many tables when we need information. In a large database, this may lead the fetch time of the queries to become very large.

It might me useful to keep some information that is often used together is the same tables when the database is very large, even though this sacrifices the tables forth normal form.

This, however, must be done with caution. Lossless join is guaranteed when the tables are on Boyce-Codd Normal Form, which all our tables are as they are on forth normal form.However, if we have tables not on BCNF, then we need to make sure our tables are secured with lossless join. If this is not the case, then we might end up accidentally creating false rows in our tables when joining them together.

3 Part 2 - Implementing the Piazza database with Java and JDBC

For this part of the project, we have implemented the Piazza database in Java with JDBC. We have used the schema that we defined in part one of this report. We have implemented functionality for the five different use cases defined by the project description.

We have chosen to use a text-based interface in our database program.

3.1 The Classes defined in our program, and what task they solve

An overview of the classes that exist in our program are found in table 17.

Classes
User
Interface
DBcon

Table 17: Classes in our database program

As seen in the table, we have implemented two classes in our program, *User* and *Interface*. The different classes each accomplish different tasks. We will now take you through the classes, and the tasks they each accomplish.

3.1.1 User

A object in the User-class is used to show whether someone is logged into the Piazza forum. A user has the attributes; *email*, *UserName* and *loggedIn*. *Email* and *UserName* are strings and give information about a user. *LoggedIn* is a boolean value and shows whether a user is in the Piazza-database. *LoggedIn* is true when a User-object is logged in. The User-class has two functionalities; `login(String loginEmail, String password, Interface db)` and `logout()`. When a user is logged in a connection between the user in the database and a User-object is established. If no user with matching email and password is found in the database, the User-object is returned empty. `logout()` sets the User-object as empty. A User-object is only nonempty if someone has successfully logged into the database. The User-Class has three constructors. The first constructor constructs an empty User-object, whereas the the last two tries to find a user in the piazza-database via the login function. The second constructor uses the constructor input to log in and in the third constructor you log in manually.

3.1.2 DBcon

The DBcon-class is used to establish a connection to the piazza database. *DBcon* has the attributes *con* and *isConnected*. *con* is the connection to the database and *isConnected* is a boolean. *isConnected* is true if *con* has been successfully connected to a database. *DBcon* has a connect- and a disconnect-function. *connect* connects *con* to a database and throws an exception if no connection is made. Likewise, disconnect disconects *con* from the database and throws an exception if unable to disconnect.

3.1.3 Interface

The Interface-Class is an extension of *DBcon*, thus it inherits the connection to the piazza-database. *Interface* is used as a way to write queries to the database and provides logic for all use cases in our program. The Interface-Class has the attribute *preStat* which is a prepared statement used to preform queries to the database. *Interface* has many different functionalities, some of them are:

- `login(String loginEmail, String password)`: Searches the database for a user with matching email and password. Returns a empty user if no match is found.
- `displayPost(int postID)`: Displays a post based on the *postID*.
- `displayThread(int threadID)`: Displays all posts in a thread based on the *threadID*.

Many of the functions are based on each other, for examplelike `displayThread` which uses `displayPost` to display all posts in a thread. Functions like `makePost`, `makeThread` and `makeFolder` creates posts, threads, folders in the database and also creates the relations between them.

3.2 An overview of the use-cases solved, and how they are realized in our program

3.2.1 How to use the database

As described later in use case 1, you cannot use any of the functionality of the database if you are not logged in and enrolled in a course. We have therefore initialized the database with one user. This user has *Email* = "test" and *Password* = "test". Therefore, fist time logging onto the database, use this, and then select the *Course* "testCourse". After this, the use case menu will give you many options. First, we advise you to generate a realization of the database. This will generate some users to the database, and enroll them in courses plus more. The different users generated are enrolled in different courses, and have different roles as students or instructors. You can now use these different users to log into the database, and try the different use-cases using different users.

The users we have are found in table 18.

Email	Password	UserName
test	test	test
bruk1@ntnu.no	101	Roger Midtstraum
bruk2@ntnu.no	202	Svein Erik Bratsberg
bruk3@ntnu.no	303	Brynjulf Owren
bruk4@ntnu.no	404	Sunniva Student
bruk5@ntnu.no	505	Simon Student
bruk6@ntnu.no	606	Sara Student
bruk7@ntnu.no	707	Helge Langseth

Table 18: A table of the users that we have made a realization for in the database

3.2.2 Use case 1

The way we have chosen to solve all use-cases in this project, is that you need to be logged into the database in order to access any of the functionality of the database. Logging in is therefore the first thing that happens when you connect to a database. The login functionality is implemented in the constructor `User(Interface db)` in the *User*-class to log in. The function asks the user to type in a email and a password. We check if this matches the email and the password of a user in the database, if no connection is made the user can try to log in again or quit. If a user is logged in the piazza forums the courses that user is enrolled in appears. A user can from there choose to enter a specific piazza forum, and by entering this forum, it will access the functionality of the forum. In this project, the functionality of the forum is mainly the other use-cases described in the project description.

3.2.3 Use case 2

In this use-case, we want a student to make a post in a folder "Exam", tagged with "Question". Here, we have written general functions that make both a student and an instructor able to start a new thread (Create the first post in a thread). This is done via the function `InputThread(courseID, term, user)`. We display the folders in the piazza forum, and the user can choose which folder to write a post in. The user can also choose to quit at this point. This is done with the function `SelectFolder(courseID, term)`. Then, the user is prompted to input the *title*, *content* and *tag* for the post, using the functions `GetTitle` and `GetContents()`. Lastly, the thread is made using the function `MakeThread(content, title, folderID, tag, user)`, inputting information about the user, and all the input given by the user. This function also displays the threadID and postID given to the newly created post, and displays the post that the user just created. If the user cancels at any point, a cancel message is displayed before the user is directed back to choosing a use-case.

3.2.4 Use case 3

For this use case, we want an instructor to reply to a post in the "Exam"-folder. Again, we have made functionality such that any user can reply to any of the threads in the piazza forum. This is done via the `InputPost(courseID, term, user)`-function. This again calls on the `selectFolder`-function so the user can select which folder to explore. From there, we input the selected folder into a `SelectThread(folderID)`-function. This works similarly to the `selectFolder`-function, by displaying all threads in a folder, and allowing the user to input which thread they want to navigate to. Then, the `DisplayThread(threadID)`-function is called. This displays all the posts in a thread. Now the user is prompted, "do they want to answer to the posts in this thread, or do they want to cancel". If the user cancels, they are taken back to the Use-case menu. If they want to answer, the `GetContents()`-function is called, and the user is prompted to input a reply to the posts. Then, the `ReplyTo(postID, content, tag, user)`-function is called, and generates the reply-post and adds it to the existing thread. The postID and the threadID of the reply is then displayed to the user. The reply made by the user is also displayed to the user. For all posts, we display if the post is made by a student or an instructor in the course.

3.2.5 Use case 4

In this use-case, a student searches for a specific keyword "WAL", the return value should be the id of all posts matching the keyword. Again, we have chosen to generalize this to a search function, where a user can search for any keyword, and the postID, contents, time the post was posted and user that wrote the post will be displayed for all posts matching the keyword. If the search word is empty, all posts in the

course will be displayed. The use-case is solved by calling the `Selectkeyword()`-function, which prompts the user to input a search word. This search word is then used in the `SearchAndDisplay(keyword, courseID, term)`-function, which again calls the functions `searchPost(keyword, courseID, term)` and `displayPost(postID)`. `SearchPost()` searches the course-piazza for all posts containing the inputted keyword, and returns the postID for all posts containing the keyword. `DisplayPost()` takes in all the postID-s returned by the `SearchPost()` function, and displays the postID and post contents to the user.

3.2.6 Use case 5

Only instructors have access to view statistics in a course. Therefore, to perform this use-case we start by checking if the user trying to access this is logged into the piazza forum as an instructor or a student. If a user is not an instructor for the course, we return a message saying that the user does not have access to this functionality.

If the user does have access to this functionality, we want to display the relevant statistics for the course.

We have written a function `DisplayStats(CourseID, term)`. This takes in the *CourseID* and *term* of the course we are interested in. This functions sends a query to the database, and compiles the relevant statistics for the course. Then it prints out the statistics, so they are easy for the instructor to read.