EBA 3420 DATABASES — TERM PAPER

Introduction **1**

1. ER-model **2**

2. Relational model **3**

3. Database creation **4**

4. Database population **6**

5. Web application **7**

Last remarks **14**

Introduction

This paper is written for the course ‘EBA 3420 Databases’ at BI Norwegian Business School. It is the major assessment weighing 70% of the final grade in the course. The day following the paper hand-in will include an exam for the remaining 30% of the final grade.

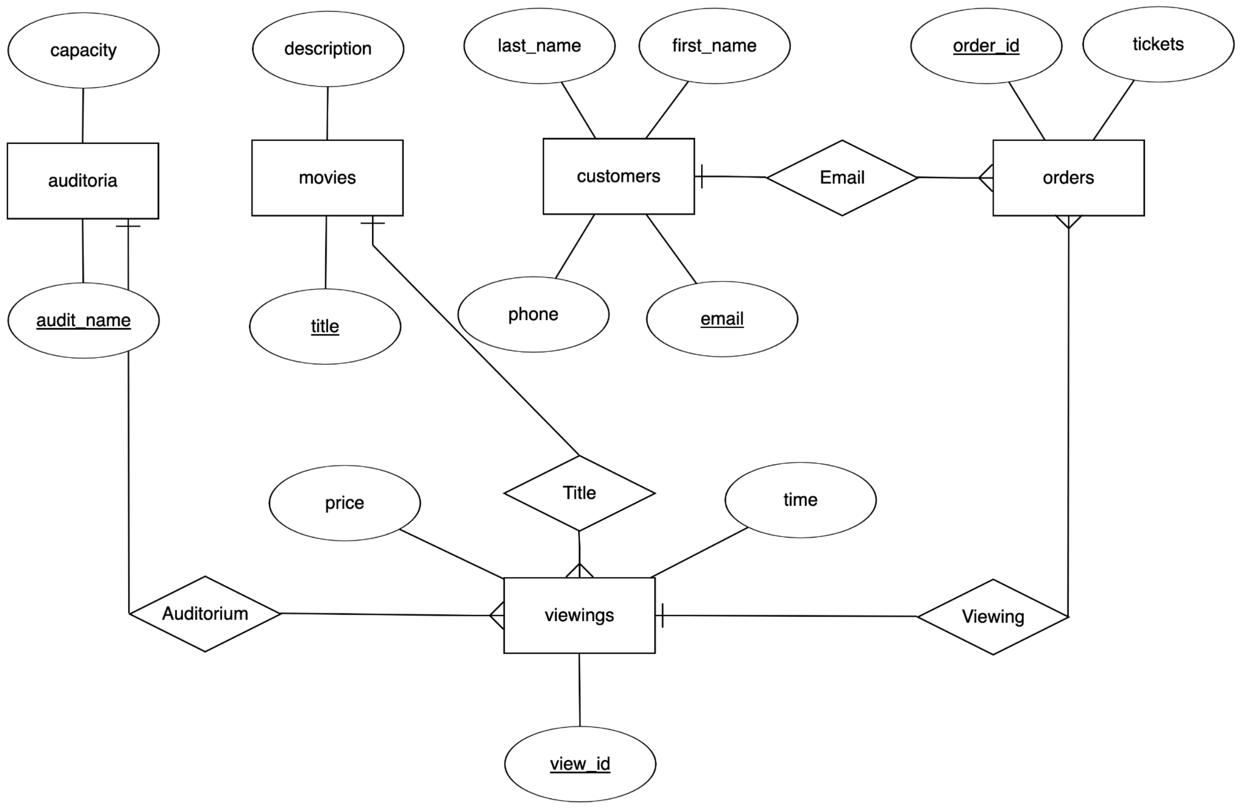
The case for this assignment is to develop a simple intranet for the employees of a small cinema-company based in Norway. The intranet needs to be an interactive web application built on the Flask framework for Python. The web application consists of two pages, where the first serves as a homepage displaying the next five viewings for the cinema, and the second serves as a detailed page displaying more information about a specific viewing. The intranet also needs to be able to order new tickets for a specific viewing.

I decided to write this paper alone based on the notion that I will learn more by doing so, and as a result may perform to a higher degree than otherwise. This paper includes detailed descriptions on how the assignment was solved, including diagrams, models and code used in the process.

Please see the attached ZIP file ‘EBA 3420.zip’ for the files requested.

1. ER-model

Relevant files: ER-model.png



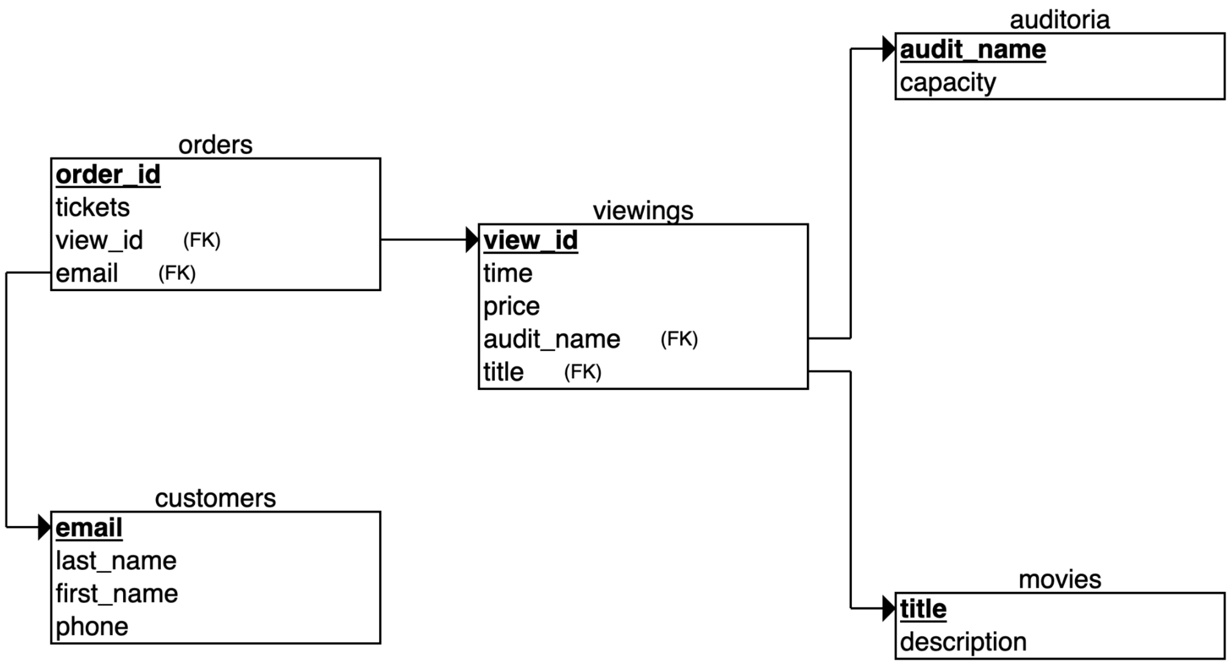
I decided to create the ER-model with the tool from ERDPlus, as it has great functionality and makes it easy to modify while maintaining a good overview.

The strategy that forms the design of this ER-model, is based on what I believe would be useful for the system as a whole. I have structured the data in a way that prevents various anomalies by first and foremost centralizing the database by the entity “viewings”.

By structuring the database in this manner, the database is protected from insertion, deletion, update and redundancy anomalies. This will be further elaborated below under the topic of the relational model.

2. Relational model

Relevant files: Relational-model.png



The relational model was made with the ERDPlus converting-tool, converting the ER-model displayed in the previous section. This was a highly effective way to control that the various keys were set correctly and were mapping to the intended locations.

A crucial factor for avoiding anomalies for the database lies within normalization. All columns use atomic values, in other words they do not contain more than one value per field. There are no nested tables as this is a relational database. Each table also has a primary key.

The way the database is structured, it is compliant to all normal forms (1NF, 2NF, 3NF), hence it is protected from various anomalies.

3. Database creation

Relevant files: new_db.py, cinematix.db, creation

To initiate the actual build of the project, a database has to be created. I created a new database “cinematix.db” with python using the sqlite3 library in the code below:

```python
import sqlite3
from sqlite3 import Error

def create_connection(db_file):
   con = None
   try:
       con = sqlite3.connect(db_file)
       print(sqlite3.version)
   except Error as e:
       print(e)
   finally:
       if con:
           con.close()

if __name__ == '__main__':
   create_connection(r"cinematix.db")
```

In the sqlite3 library, when you attempt to make a connection to a database that does not exist, it will create one for you in the location where the python file is stored and name it as the string that was used as a reference for the connection.

I decided to approach it in this manner, as it was a simple way to ensure that the database created ended up in the same folder as the other files for the project.

With the database up and running, the tables for it were ready to be created.

```sql
CREATE TABLE customers
(
 last_name VARCHAR,
 first_name VARCHAR,
 phone INT,
 email VARCHAR,
 PRIMARY KEY (email)
);
CREATE TABLE auditoria
(
 capacity INT,
 audit_name INT,
 PRIMARY KEY (audit_name)
);
CREATE TABLE movies
(
 descript VARCHAR,
 title VARCHAR,
 PRIMARY KEY (title)
);
CREATE TABLE orders
(
 order_id INT,
 tickets INT,
 view_id INT,
 email VARCHAR,
 PRIMARY KEY (order_id),
 FOREIGN KEY (view_id) REFERENCES viewings(view_id),
 FOREIGN KEY (email) REFERENCES customers(email)
);
CREATE TABLE viewings
(
 view_id INT,
 title VARCHAR,
 audit_name VARCHAR,
 date_time DATE,
 price INT,
 PRIMARY KEY (view_id),
 FOREIGN KEY (audit_name) REFERENCES auditoria(audit_name)
);
```

I decided to write the SQL code manually as a principle of quality control and as recommended in the hand-out. The ERDPlus-tool for converting the relational model into SQL was however helpful to make sure I did not leave anything out, as well as controlling for syntax errors.

4. Database population

Relevant files: population.db, cinematix.db, CSV[folder]

Now that the database and tables are created, it is time to populate the database. The approach I took for preparing data to populate the database with, was to first create it in spreadsheets (using 'Google sheets'). I made one sheet for each table in the database and filled it in with arbitrary data. The different sheets could then be downloaded as CSV files.

The CSV files could then be inserted into the different tables by running the following Python script.

```python
import pandas as pd
import sqlite3

path = '/Users/martinfuglset/EBA3420'

con = sqlite3.connect(path + '/Database/cinematix.db')
cursor = con.cursor()

def import_csv(name):
   cursor.execute('DELETE FROM '+name+';')

   df = pd.DataFrame(pd.read_csv(path + "/CSV/eba3420 - " + name + ".csv"))
   rows = len(df)

   string = '''INSERT INTO '''+ name +'''
   VALUES
   '''

   for row in df.itertuples():
       string += str(row[1:rows])
       string += ","

   string = string[:-1] + ";"
   cursor.execute(string)

auditoria = import_csv("auditoria")
customers = import_csv("customers")
movies = import_csv("movies")
orders = import_csv("orders")
viewings = import_csv("viewings")

con.commit()
con.close()
```

The function is very useful for testing as it initially clears the table before adding new data.

5. Web application

Relevant files: cinematix.db, cinematix.py, templates[folder]

Homepage

The assignment required this project to be made with the Flask framework for python. Naturally, the process started with importing the needed libraries and packages into the newly created python file 'cinematix.py'.

```
from flask import Flask, request, render_template
import pandas as pd
import sqlite3

pd.options.display.max_colwidth = 200
```

I noticed during testing that long strings could be "cut off", and figured out that this was because of the default maximum display for column width, which explains why the code includes an adjustment of this setting (200 was an arbitrary value that worked for the strings used in this project).

```
con = sqlite3.connect('/Users/martinfuglset/EBA3420/Database/cinematix.db')
base_url = "http://127.0.0.1:5000"

query = """
SELECT viewings.view_id, viewings.title, movies.descript, viewings.audit_name,
auditoria.capacity-sum(orders.tickets) AS capacity_left, viewings.date_time,
viewings.price
FROM viewings
INNER JOIN orders ON viewings.view_id=orders.view_id
INNER JOIN auditoria ON viewings.audit_name=auditoria.audit_name
INNER JOIN movies ON viewings.title=movies.title
"""

df_home = pd.read_sql(query + "GROUP BY viewings.view_id order by date_time asc limit
5;", con)
```

In order to interact with the database created earlier, I created the variable 'con' which stores the connection through the 'connect' function in the 'sqlite3' library. The default setting for running projects in Flask is set to local hosting with port 5000. This is stored as the variable 'base_url' for several practical purposes. Though the database is created and populated, it still needs to be formatted for the project. A query string, simply stored as the variable 'query', was created for this purpose. The string selects the columns needed from the various tables, then joins the tables needed on the 'viewings' table.

Now that the query is in place, a dataframe can be created for the table that is ought to be displayed on the homepage. This is stored in the variable ‘df_home’ and creates a dataframe using pandas, and adding the end of the query string for the specific use. The reason this string is not stored in the ‘query’ variable, is that the string stored in the ‘query’ variable will come to use elsewhere in the code.

Up next was to create a function for the table on the homepage to be converted into HTML.

```
def to_html_table(pandas_table):
   html = '<table border=1>'
   html += '<th>Title</th><th>Date and Time</th><th>Auditorium</th><th>Ticket
price</th>'

   for row in pandas_table.itertuples():
       html += '<tr>'
       html += '<td>' + str(row[2]) + '</td>'
       html += '<td>' + str(row[6]) + '</td>'
       html += '<td>' + str(row[4]) + '</td>'
       html += '<td>' + str(row[7]) + '</td>'
       id = str(row[1])
       html += """<td><a href='"""+ base_url +"""/viewing?id="""+id+"'"""">more
info</a></td>"""
       html += '</tr>'
   html += '</table>'

   return html

table_home = to_html_table(df_home)

viewing_list = df_home['view_id'].to_list()
```

As the format would not change depending on the data, the structure was more or less hardcoded, but naturally using variables for placing data in the table as the data is subject to change. As you can see in the query string previously shown, the for loop references rows from specific columns according to the query so that it displays only the information requested. The HTML code for the table gets stored in the variable ‘table_home’, which contains a string with the HTML code. The variable ‘viewing_list’ stores a list containing all valid IDs for the viewings, which has multiple practical applications. In the lower end of the function, there is a substring added to the main string containing links. The intended route for these links were ‘/viewing’ followed by a query string with the viewing id for the viewing that contained the link that was clicked.

As recommended in web app development, a good practice is to structure the application using HTML templates. You will find the template for the home page below.

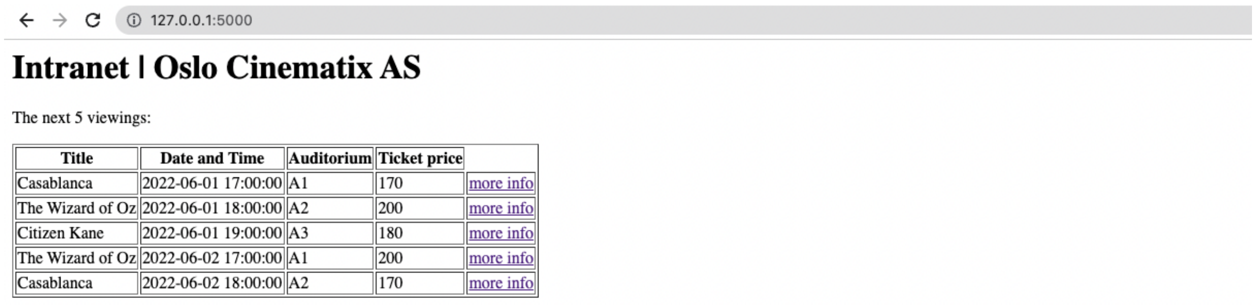
```
<html>
        <head></head>
        <body>
                <h1>Intranet | Oslo Cinematix AS</h1>
                <p>The next 5 viewings:</p>
        </body>
</html>
```

The template for the homepage is rather scarce. But I decided to use it anyway as it contributes to a stronger foundation for the web app, assuming the web app can be a subject to further development. As you can see in the template, the table for the homepage is nowhere to be seen. That is because the 'table_home' variable gets added in the function for the homepage.

All data is prepared and ready to be deployed. The next step is to initiate the web application. Flask has made this task extremely easy, and is simply done by writing '`app = Flask__name__)`'.
Next, we need a route that the page sends us to. The homepage is naturally assigned to just "/" in most pages, and that is what this web app will do as well.

```
app = Flask(__name__)

@app.route("/")
def home():

   return render_template('home.html') + table_home
```

The function assigned to the homepage route is rather simple, and is returning the rendered HTML-template followed by 'table_home', both displayed earlier in the paper. These values are both strings in proper HTML-syntax. What the function returns is a string that is sent to the assigned route.



127.0.0.1:5000

Intranet | Oslo Cinematix AS

The next 5 viewings:

Title	Date and Time	Auditorium	Ticket price	
Casablanca	2022-06-01 17:00:00	A1	170	more info
The Wizard of Oz	2022-06-01 18:00:00	A2	200	more info
Citizen Kane	2022-06-01 19:00:00	A3	180	more info
The Wizard of Oz	2022-06-02 17:00:00	A1	200	more info
Casablanca	2022-06-02 18:00:00	A2	170	more info

This is the result when visiting the homepage URL. It displays exactly the information requested in the project description.

Detailed page

Now that the homepage is up and running, the “more info”-links need detailed pages.

Similar to the query for the homepage, a query string for the order table has to be made.

```
orders_table = """
SELECT customers.last_name as 'Last name', customers.first_name as 'First name',
orders.email AS 'Email', customers.phone as 'Phone nr.', sum(orders.tickets) as
'Tickets ordered'
FROM orders
INNER JOIN customers ON orders.email=customers.email
"""

def datframe(id):
   con = sqlite3.connect('/Users/martinfuglset/EBA3420/Database/cinematix.db')
   df = pd.read_sql(orders_table + "WHERE view_id = " + str(id) + " GROUP BY
orders.email order by last_name;", con)
   return df


def get_movie_data(id: str):
   con = sqlite3.connect('/Users/martinfuglset/EBA3420/Database/cinematix.db')
   df = pd.read_sql(query + "WHERE viewings.view_id = " + str(id) + ";", con)
   return df


return_link = "<a href="+ base_url +">Return home</a>"

email_list = pd.read_sql('SELECT email FROM customers',con)['email'].values.tolist()
```

The functions create useful dataframes for the detailed page.

HTML template for detailed page.

```html
<html>
       <head></head>
       <body>
           <h1>More information</h1>

           <p>Title: {{title}}</p>
           <p>Time: {{time}}</p>
           <p>Auditorium: {{audit}}</p>
           <p>Ticket price: {{price}}</p>
           <p>Capacity left: {{capacity}}</p>
           <p>Description: {{descript}}</p>

           <p>Ordered tickets: </p>
       </body>
</html>
```

HTML template for order form.

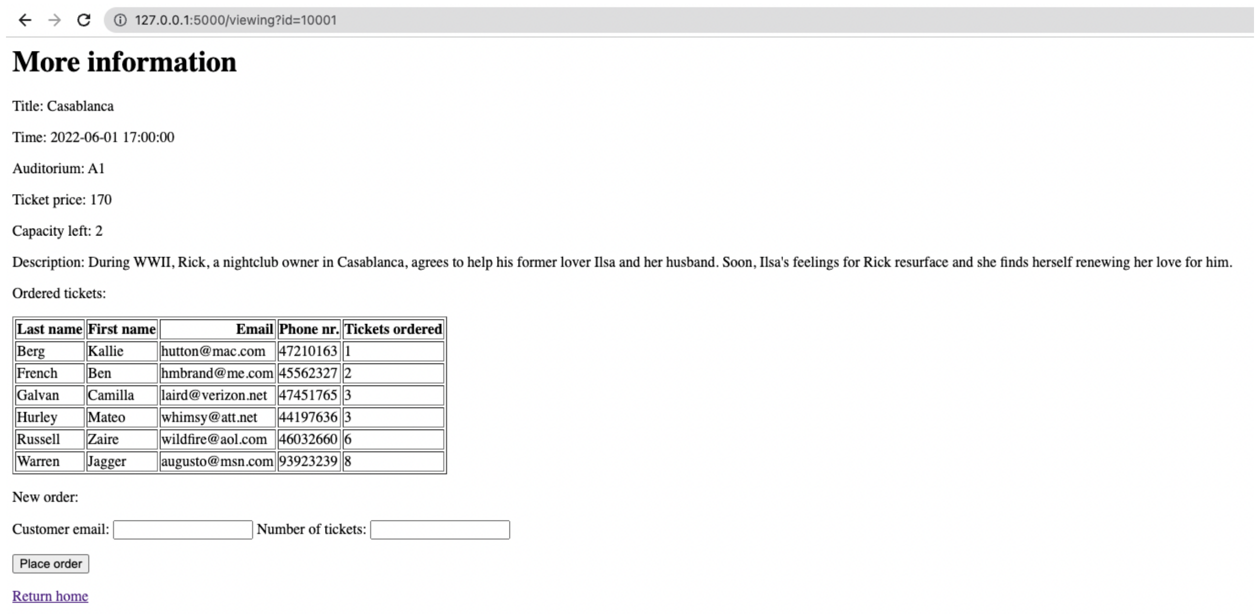
```html
<html>
       <head></head>
       <body>
           <p>New order:</p>

           <form action="#" method="POST">
               <p>Customer email:
               <input type="text" name="em">
               Number of tickets:
               <input type="integer" name="numb">
               </p>
               <p>
               <input type="submit" value="Place order">
               </p>
               </form>
       </body>
</html>
```

I decided to separate this from the other template as this provides a more dynamic approach for the table of ordered tickets.

App route for detailed page.

```
@app.route("/viewing", methods=['POST','GET'])
def page():
   id = request.args.get('id')
   if int(id) not in viewing_list:
       return "<p>You have entered an invalid URL.</p>" + return_link
   pass
   movie_data = get_movie_data(id)
   movie_data = movie_data.to_dict()
   title = movie_data['title'][0]
   descript = movie_data['descript'][0]
   time = movie_data['date_time'][0]
   audit = movie_data['audit_name'][0]
   price = movie_data['price'][0]
   capacity = movie_data['capacity_left'][0]
   table = datframe(id).to_html(index=False)

   render = render_template('more_info.html', title=title, time=time, audit=audit,
price=price, capacity=capacity, descript=descript, base_url=base_url) + table +
render_template('order_form.html')

   if request.method == 'POST':
       eml = request.form["em"]
       numb = request.form["numb"]

       if eml in email_list and int(numb) <= capacity:

           string = """
           INSERT INTO orders
           VALUES
           (
               (SELECT MAX(order_id) FROM orders) + 1,"""+ numb +""","""+ id
+""",'"""+ eml +"""')
           ;
           """
           con = sqlite3.connect('/Users/martinfuglset/EBA3420/Database/cinematix.db')
           cursor=con.cursor()
           cursor.execute(string)
           con.commit()
           return render + "<p>" + numb +" tickets ordered for "+ eml +"</p>" +
return_link
       else:
           return render + "<p>ERROR: email not in system or not enough tickets
available</p>" + return_link

   else:
       return render + return_link
```

Arguably the most crucial part of this code is the 'request' module. For the web app to fulfill the project description, it needs to have a functional form that can order tickets (given sufficient capacity). The request module provides a simple way to extract the desired values for updating the database.


127.0.0.1:5000/viewing?id=10001

More information

Title: Casablanca

Time: 2022-06-01 17:00:00

Auditorium: A1

Ticket price: 170

Capacity left: 2

Description: During WWII, Rick, a nightclub owner in Casablanca, agrees to help his former lover Ilsa and her husband. Soon, Ilsa's feelings for Rick resurface and she finds herself renewing her love for him.

Ordered tickets:

Last name	First name	Email	Phone nr.	Tickets ordered
Berg	Kallie	hutton@mac.com	47210163	1
French	Ben	hmbrand@me.com	45562327	2
Galvan	Camilla	laird@verizon.net	47451765	3
Hurley	Mateo	whimsy@att.net	44197636	3
Russell	Zaire	wildfire@aol.com	46032660	6
Warren	Jagger	augusto@msn.com	93923239	8

New order:

Customer email: Number of tickets:

When visiting the URLs for the various viewings, this is what is displayed. The page is fully functional according to the project description.

Error handling, running and closing connection

It is practical to include an error handler for invalid routes. This lets the user know that the page does not exist, and will provide a link to go back to the homepage.

```
@app.errorhandler(404)
def invalid_route(e):
   return "<p>You have entered an invalid URL.</p>" + return_link

if __name__ == "__main__":
   app.run()

con.close()
```

Now everything is in place to run the application, which is done by running:

'`if __name__ == "__main__": app.run()`'

It is considered good practice to close the database connection at the end of the program. This will only affect transactions that are still open.

Last remarks

Process

My approach consisted in large part of attempting to create something that fully achieves the project description, while being designed in a way that is dynamic and prevents error when edited.

Summarized, I am very satisfied with the outcome. Though it was challenging, I am confident that the project was solved to a satisfactional degree despite writing it single-handedly. The project was all-in-all a positive learning experience, and I have come to a greater understanding applying the theory from the course into a more pragmatic context.

Data

Movie titles are derived from actual movies arbitrarily chosen. The description for each movie is derived from Google Knowledge Panels. Names and emails were gathered from an online generator: https://www.randomlists.com/. All other data was made up.

Code

I decided to leave out comments in the code as I think the redundancy of both comments in the code in addition to the explanation in the paper seemed redundant and cluttered. I would most certainly have included comments if this was a real world project.