Project Design Report

CS 353 - Section 2 - Group 23

Ali İlteriş Tabak

Pınar Göktepe

Süleyman Can Özülkü

Zülâl Bingöl

4.4.2016

http://ilteristabak.github.io/movierentalsystem/

Table of Contents

Table of Figures

1. Problem Statement

In this project, it is intended to prepare a movie rental data store management system via a web application. It is basically designed to enable the customers rent their desired movies within limited period of time or buy them. The movies are sold or rented in Blu-ray and DVD format.

The system stores different kinds of copies of movies and detailed information about them such as directors, artists and genre of the movies. A customer can first become a member of the web application and makes a log in to the system with a password. Customers can search for the movies directly by movie names or can use the filtrations that the system supports if there is no specific movie in mind. The control mechanism of the system prevents the customers from buying or renting the movies according to the age limits. The customer can either rent the movies for period of time and pay the rent or buy the movies. The rental penalties in terms of extra money are applied to the customers with their late returns. The system provides the customers with the opportunity of requesting the order of movies which do not exist in the stocks of the system.

Data store management system keeps the records of all the customers and renting information. By this way, the admin of the web application can maintain the control of the renting logs and perform the necessary actions on the customer accounts such as prohibiting the accounts which do not pay their debts. The system can also apply discounts and promotions on the movies, based on the renting history of the customers.

Additionally, the system stores the data of the employees and the distributors. By this way, the admin keep track of the administration via the web application.

2. Revised Parts of Requirements Report

While revising the system, some alterations have been performed in order to improve the relations and the entities in terms of practicality and efficiency. According to these alterations and the feedback given by the teaching assistant for the requirements report, the following changes were applied to the E/R Diagram of the system;

Movie:

- "copy_type" has been removed and moved to Stock entity.
- "price" attribute has been customized into "price_rent" and "price_buy".

Genre:

Genre is not a weak entity anymore.

- "genre_id" is made to be the primary key.

Subject:

Subject used to contain total participation from both Movie side and the Genre side. In the new system, Subject represents many-to-many relationship between Movie and Genre.

Stock:

Stock is weak entity in the new system.

- "stock_id" is the weak primary key of Stock entity.
- "copy_type" attribute has been added in order to keep the type of the movies.
- "rent_count" attribute has been added to keep the count of the maximum number of movies reserved for rental. Every time someone rents a movie, the rent_count of that specific movie decrements by one.
- "copy_number" attribute now represents the number of copies of a specific movie, reserved for sale.

Has:

Has relation now contains total participation from Stock side; if a movie is removed from the database then the corresponding stock is also removed.

Distributor:

- "address" attribute has been added to the system.
- "password" attribute has been added to provide the Distributor login to the system.
- “valid” attribute has been added for system admin to check the validity of the distributor.

RentalLog & Records: Completely removed from the system.

Aggregation relationship of Stock-Rent-Customer: Removed from the system.

Customer:

- "promotion" attribute has been removed from this entity.
- "buy_count" attribute has been added to keep the count of the total number of movies bought by the customer.
- "rent_count" attribute has been added to keep the count of the movies rented by the customer.
- "favourite" attribute has been removed.

Rent:

Rent relation used to be between the Stock and the Customer. In the new system, Rent relation is held between Movie and the Customer entities.

- "rent_date" attribute has been added to keep the rental date. This is one of the primary keys of the Rent relation.
- "return_date" has been added to the system to check the return date.

Promotion:

Promotion is a new entity added to the system.

- "promotion_type" attribute is the primary key of the promotion entity.

Given:

Given relation has been added the system. It represents the many-to-many relation between Promotion and Customer.

OrderOfNewProducts:

- "movie_name" attribute has been added to system to keep the name of the requested movie.
- "movie_year" attribute has been added to system to keep the year of the requested movie.
- "movie_type" attribute has been added to system to keep the type of the requested movie.

Order:

- "status" attribute has been declared for status of the Order.

Buy:

Buy relation used to be between the Stock and the Customer. In the new system, Buy relation is held between Movie and the Customer entities.

Artist:

- "role" attribute has been declared to keep the role of the artist.

Director:

- "alma_mater" attribute has been declared for alma_mater of the director.

Award:

- "year" attribute has been declared.

Won_M:

The name of the relation has been changed from "Won" to "Won_M". The cardinality of the relation has been changed to be "many" on Award side.

Won_A:

This is a newly declared relation. It holds the award-winning relation between the Artist and the Award. The cardinality is "many" on Award's side and "one" on Artist's side meaning that an artist can have more than one Award but a specific award goes to only one Artist.

Won_D:

This is a newly declared relation. It holds the award-winning relation between the Director and the Award. The cardinality is "many" on Award's side and "one" on Director's side meaning that a director can have more than one Award but a specific award goes only to one Director.

3. Relational Schemas

3.1 Person

Relational Model:

person (person_id, person_name, age)

Functional Dependencies:

person_id -> person_name age

Candidate Keys:

{(person_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE person (
        person_id       int PRIMARY KEY AUTO_INCREMENT,
        person_name     varchar(40) NOT NULL,
        age             int NOT NULL) ENGINE = InnoDB;
```

3.2 User

Relational Model:

user (user_id, authorization, e_mail, password, address, phone)

Functional Dependencies:

user_id -> authorization e_mail, password address phone

Candidate Keys:

{(user_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE user (
        user_id         int PRIMARY KEY,
        FOREIGN KEY (user_id) references person (person_id),
        authorization   int NOT NULL,
        e_mail          varchar(40) NOT NULL,
        password        varchar(40) NOT NULL,
        address         varchar(40),
        phone           varchar(40) ) ENGINE=InnoDB;
```

3.3 Employee

Relational Model:

employee (employee_id, registery_number)

Functional Dependencies:

employee _id -> registery_number

Candidate Keys:

{(employee _id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE employee (
        employee_id             int PRIMARY KEY,
        FOREIGN KEY (employee _id) references person (person_id),
        registery_number        int AUTO_INCREMENT) ENGINE=InnoDB;
```

3.4 Director

Relational Model:

director (director_id, alma_mater)

Functional Dependencies:

director_id -> alma_mater

Candidate Keys:

{(director_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE director (
        director_id      int PRIMARY KEY,
        FOREIGN KEY (director_id) references person (person_id),
        alma_mater     varchar(40) NOT NULL ) ENGINE=InnoDB;
```

3.5 OrderOfNewProducts

Relational Model:

orderOfNewProducts(order_id, cost, movie_name, movie_year, movie_type)

Functional Dependencies:

order_id -> cost movie_name movie_year movie_type

Candidate Keys:

{(order_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE orderOfNewProducts (
        order_id        int PRIMARY KEY AUTO_INCREMENT,
        cost            float NOT NULL,
        moive_name      varchar(45) NOT NULL,
        movie_year      DATE NOT NULL,
        movie_type      varchar(40) NOT NULL ) ENGINE=InnoDB;
```

3.6 Order

Relational Model:

order (order_id, distributor_id, status)

Functional Dependencies:

order_id distributor_id -> status

Candidate Keys:

{(order_id, distributor_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE order (
        order_id        int NOT NULL,
        distributor_id  int,
        status          int NOT NULL,
        PRIMARY KEY (order_id, distributor_id),
        FOREIGN KEY (order_id) references orderOfNewProducts,
        FOREIGN KEY (distributor_id) references distributor) ENGINE=InnoDB;
```

3.7 Subject

Relational Model:

subject (movie_id, genre_id)

Functional Dependencies:

No dependencies.

Candidate Keys:

{(movie_id, genre_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE subject (
        movie_id        int NOT NULL,
        genre_id        int NOT NULL,
        PRIMARY KEY (movie_id, genre_id),
        FOREIGN KEY (movie_id) references movie,
        FOREIGN KEY (genre_id) references genre) ENGINE=InnoDB;
```

3.8 Won_M

Relational Model:

won_M (movie_id, award_id)

Functional Dependencies:

No dependencies.

Candidate Keys:

{(movie_id, award_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE won_M (
        movie_id        int NOT NULL,
        award_id        int NOT NULL,
        PRIMARY KEY (movie_id, award _id),
        FOREIGN KEY (movie_id) references movie,
        FOREIGN KEY (award _id) references award) ENGINE=InnoDB;
```

3.9 Won_A

Relational Model:

won_A (artist_id, award_id)

Functional Dependencies:

No dependencies.

Candidate Keys:

{(artist_id, award_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE won_A (
        artist_id       int NOT NULL,
        award_id        int NOT NULL,
        PRIMARY KEY (artist _id, award _id),
        FOREIGN KEY (artist _id) references artist,
        FOREIGN KEY (award _id) references award) ENGINE=InnoDB;
```

3.10 Won_D

Relational Model:

won_D (director _id, award_id)

Functional Dependencies:

No dependencies.

Candidate Keys:

{(director_id, award_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE won_D (
        director_id     int NOT NULL,
        award_id        int NOT NULL,
        PRIMARY KEY (director_id, award_id),
        FOREIGN KEY (director_id) references director,
        FOREIGN KEY (award_id) references award) ENGINE=InnoDB;
```

3.11 Play

Relational Model:

play (movie_id, artist _id)

Functional Dependencies:

No dependencies.

Candidate Keys:

{(movie_id, artist _id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE play (
        movie_id        int NOT NULL,
        artist_id       int NOT NULL,
        PRIMARY KEY (movie _id, artist_id),
        FOREIGN KEY (movie _id) references movie,
        FOREIGN KEY (artist _id) references artist) ENGINE=InnoDB;
```

3.12 Direct

Relational Model:

direct (movie_id, director _id)

Functional Dependencies:

No dependencies.

Candidate Keys:

{(movie_id, director _id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE direct (
        movie_id        int NOT NULL,
        director_id     int NOT NULL,
        PRIMARY KEY (movie_id, director_id),
        FOREIGN KEY (movie_id) references movie,
        FOREIGN KEY (director_id) references director) ENGINE=InnoDB;
```

3.13 Approve

Relational Model:

approve (employee_id, customer_id, order_id)

Functional Dependencies:

No dependencies.

Candidate Keys:

{(employee_id, customer_id, order_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE approve (
        employee_id         int NOT NULL,
        customer_id         int NOT NULL,
        order_id            int NOT NULL,
        PRIMARY KEY (employee_id, customer _id, order_id),
        FOREIGN KEY (employee_id) references employee,
        FOREIGN KEY (customer_id) references customer,
        FOREIGN KEY (order _id) references orderOfNewProducts) ENGINE=InnoDB;
```

3.14 Award

Relational Model:

award(award_id, award_name, award_type, year)

Functional Dependencies:

award_id -> award_name award_type year

Candidate Keys:

{{award_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE award (
        award_id        int PRIMARY KEY AUTO_INCREMENT,
        award_name      varchar(40) NOT NULL,
        award_type      varchar(40) NOT NULL,
        year            int
        ) ENGINE=InnoDB;
```

3.15 Genre

Relational Model:

genre(genre_id, genre_name)

Functional Dependencies:

genre_id -> genre_name

Candidate Keys:

{{genre_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE genre (
        genre_id        int PRIMARY KEY,
        genre_name      varchar(45) NOT NULL) ENGINE=InnoDB;
```

3.16 Movie

Relational Model:

movie (movie_id, movie_name, age_limitation, release_year, price_rent, price_buy)

Functional Dependencies:

movie_id -> movie_name age_limitation release_year price_rent price_buy

Candidate Keys:

{{movie_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE movie (
        movie_id        int PRIMARY KEY AUTO_INCREMENT,
        movie_name      varchar(45) NOT NULL,
        age_limitation  varchar(45) NOT NULL,
        release_year    date NOT NULL,
        price_rent      int NOT NULL,
        price_buy       int NOT NULL
        ) ENGINE=InnoDB;
```

3.17 Artist

Relational Model:

artist (artist_id, role)

Functional Dependencies:

artist_id -> role

Candidate Keys:

{{artist_id}}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE artist (
        artist_id       int PRIMARY KEY,
        role            varchar(45) NOT NULL,
        FOREIGN KEY (artist_id) references person (person_id),
        ) ENGINE=InnoDB;
```

3.18 Distributor

Relational Model:

distributor (disributor_id, distributor_name, e_mail, phone, address, password, valid)

Functional Dependencies:

distibutor_id -> distributor_name e_mail phone

Candidate Keys:

{(distibutor_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE distributor (
        distributor_id          int PRIMARY KEY AUTO_INCREMENT,
        distributor_name        varchar(40) NOT NULL,
        e_mail                  varchar(40),
        phone                   int,
        address                   varchar(40),
        password                  varchar(40) NOT NULL,
        valid                     int NOT NULL
        )  ENGINE = InnoDB;
```

3.19 Promotion

Relational Model:

promotion (promotion_type)

Functional Dependencies:

No dependency

Candidate Keys:

{(promotion_type)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE promotion (
        promotion_type          int PRIMARY KEY
        )  ENGINE = InnoDB;
```

3.20 Customer

Relational Model:

customer (customer_id, current_debt, paid_debt, buy_count, rent_count)

Functional Dependencies:

customer_id -> current_debt paid_debt buy_count, rent_count

Candidate Keys:

{(customer_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE customer (
        customer_id                 int PRIMARY KEY,
        current_debt                int,
        paid_debt                   int ,
        buy_count                   int,
        rent_count                  int,
        FOREIGN KEY (customer _id) references person (person_id)
        ) ENGINE = InnoDB;
```

3.21 Stock

Relational Model:

stock (stock_id, copy_number, copy_type, rent_count)

Functional Dependencies:

stock_id -> copy_number copy_type rent_count

Candidate Keys:

{(stock_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE stock (
        stock_id            int PRIMARY KEY,
        copy_number         int NOT NULL,
        copy_type           varchar(40) NOT NULL ,
        rent_count          int AUTO_DECREMENT,
        FOREIGN KEY (stock_id) references movie (movie_id),
        ON DELETE CASCADE) ENGINE = InnoDB;
```

3.22 Buy

Relational Model:

buy (movie_id, customer _id)

Functional Dependencies:

No dependency

Candidate Keys:

{(movie_id, customer_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE buy (
        movie_id            int NOT NULL,
        customer_id         int NOT NULL,
        PRIMARY KEY (movie_id, customer_id),
        FOREIGN KEY (movie_id) references movie,
        FOREIGN KEY (customer_id) references customer) ENGINE = InnoDB;
```

3.23 Rent

Relational Model:

rent (movie_id, customer _id, rent_date, return_date)

Functional Dependencies:

rent_date -> return_date

Candidate Keys:

{(movie_id, customer_id, rent_date)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE rent (
        movie_id            int NOT NULL,
        customer_id         int NOT NULL,
        rent_date           date NOT NULL,
        return_date         date NOT NULL,
        PRIMARY KEY (movie_id, customer _id, rent_date),
        FOREIGN KEY (movie_id) references movie,
        FOREIGN KEY (customer_id) references customer) ENGINE = InnoDB;
```

3.24 Provide

Relational Model:

provide (distributor_id, stock_id)

Functional Dependencies:

No dependency

Candidate Keys:

{(distributor_id, stock_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE provide (
        distributor_id          int NOT NULL,
        stock_id                int NOT NULL,
        PRIMARY KEY (distributor_id, stock_id),
        FOREIGN KEY (distributor_id) references distributor,
        FOREIGN KEY (stock_id) references stock) ENGINE = InnoDB;
```

3.25 Request

Relational Model:

request (customer_id, order_id)

Functional Dependencies:

No dependency

Candidate Keys:

{(customer_id, order_id)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE request (
        customer_id         int NOT NULL,
        order_id            int NOT NULL,
        PRIMARY KEY (customer_id, order_id),
        FOREIGN KEY (customer_id) references customer,
        FOREIGN KEY (order_id) references orderOfNewProducts) ENGINE = InnoDB;
```

3.26 Given

Relational Model:

given (customer_id, promotion_type)

Functional Dependencies:

No dependency

Candidate Keys:

{(customer_id, promotion_type)}

Normal Form:

BCNF

Table Definition:

```
CREATE TABLE request (
        customer_id         int NOT NULL,
        promotion_type      varchar(40) NOT NULL,
        PRIMARY KEY (customer_id, promotion_type),
        FOREIGN KEY (customer_id) references customer,
        FOREIGN KEY (promotion_type) references promotion) ENGINE = InnoDB;
```

4. Functional Dependencies and Normalization of Tables

All tables are normalized and reorganized according to Boyce-Codd Normal Form (BCNF) and their keys and functional dependencies are stated in the previous section (Final List of Tables) for each one of the relations.

5. Functional Components

5.1 Use Cases / Scenarios

In Movie Rental Store Date Management System, there are four different types of users as customer, employee, distributor and system admin. For each user, the system has different functionalities and several common abilities. In order to use the system, each user must register and login to the system. After the login stage, there are some limitations varying on the type of the user.

5.1.1 Customer

- Customer can create a customer account with username, age, email and password.
- Customer can login to the system using his username and password.
- Customer can view his/her own profile which includes all the history about movie rentals, information about bought movies. Moreover, customer can see his/her current debt and paid debt.
- Customer can change his/her information about password, address, and phone.
- Customer can search for a movie by movie name.
- Customer can search a genre to list all movies of it in the system.
- Customer can search for a director by director name to list the all his/her movies in the system.
- Customer can search for an artist by artist name to list the all his/her movies in the system.
- Customer can search for a movie specifying the award and year.
- Customer can search for an artist specifying the award and year.
- Customer can search for a director specifying the award and year.
- Customer can list information about a movie.
- Customer can rent a movie by specifying the medium type (Blu-ray or DVD).
- Customer can buy a movie by specifying the medium type (Blu-ray or DVD).
- Customer can pay the rental price.
- Customer can pay the price to buy.
- Customer can request an order of a new product.

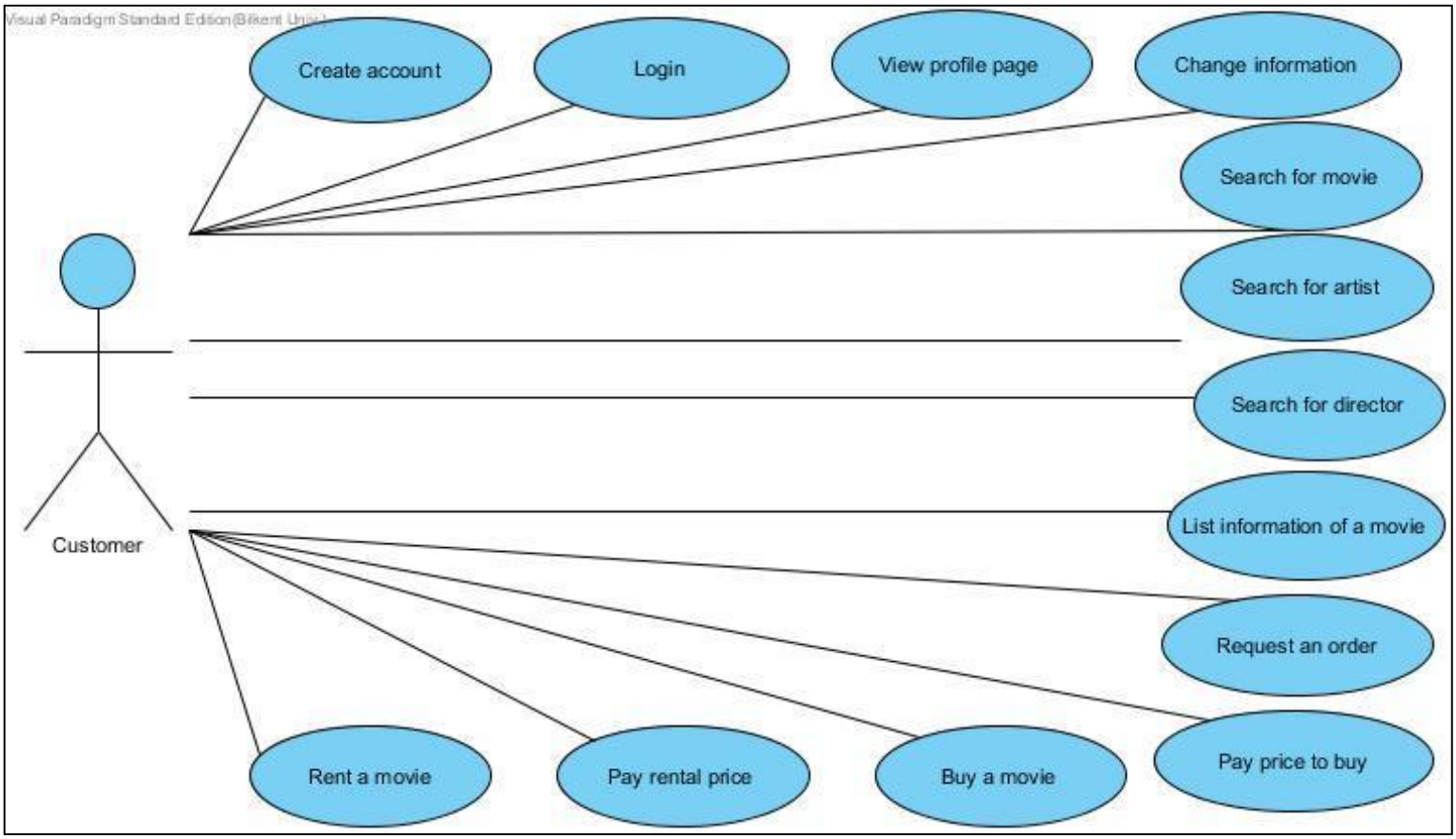

Figure 1: Customer Use Case

5.1.2 Employee

- Employee can create an employee account with user name, email and password.
- Employee can login to the system using his user name and password.
- Employee can view his/her own profile which includes his/her name, age, registry number, authorization, email, address and phone.
- Employee can change his/her information about password, address and phone.
- Employee can view the orders made by customers. The cost of the orders can be seen by employee. Also, employee can see the name, email and phone of the customer who made the order.
- Employee can approve/disapprove these orders.
- Employee can list the customers and view their current debts.
- Employee can set promotion details according to current status of the customers.

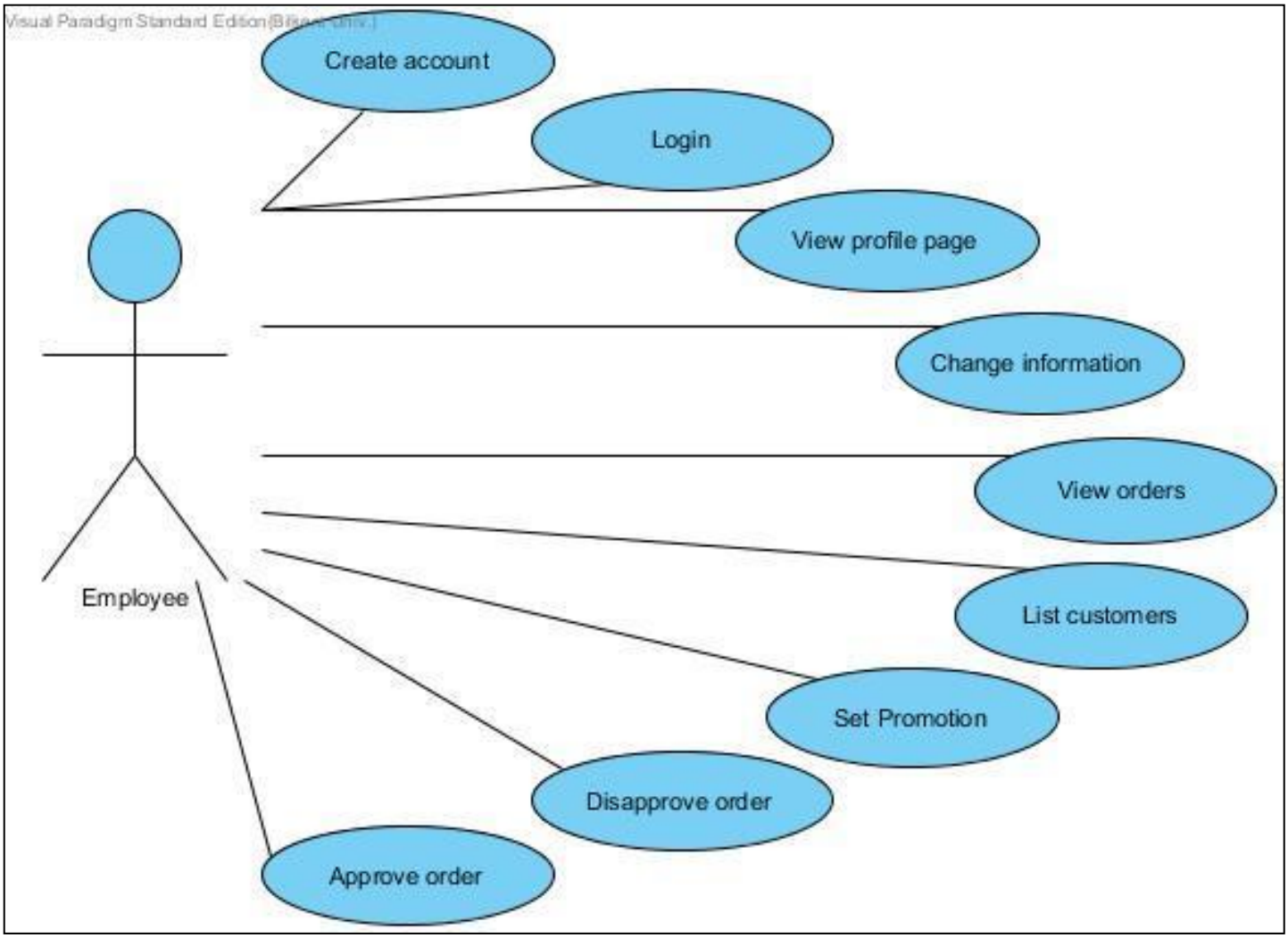

Figure 2: Employee Use Case

5.1.3 Distributor

- Distributor can create a distributor account with username, email and password.
- If System Admin approves the account of distributor, Distributor will be able to do followings:
    - Distributor can login to the system using his/her username and password.
    - Distributor can view his/her own profile which includes his/her name, email, and phone.
    - Distributor can change his/her information about password and phone
    - Distributor can view ordered movies that are requested by Customers and approved by Employee.
    - Distributor can choose an order to provide which is not chosen by any other distributor.

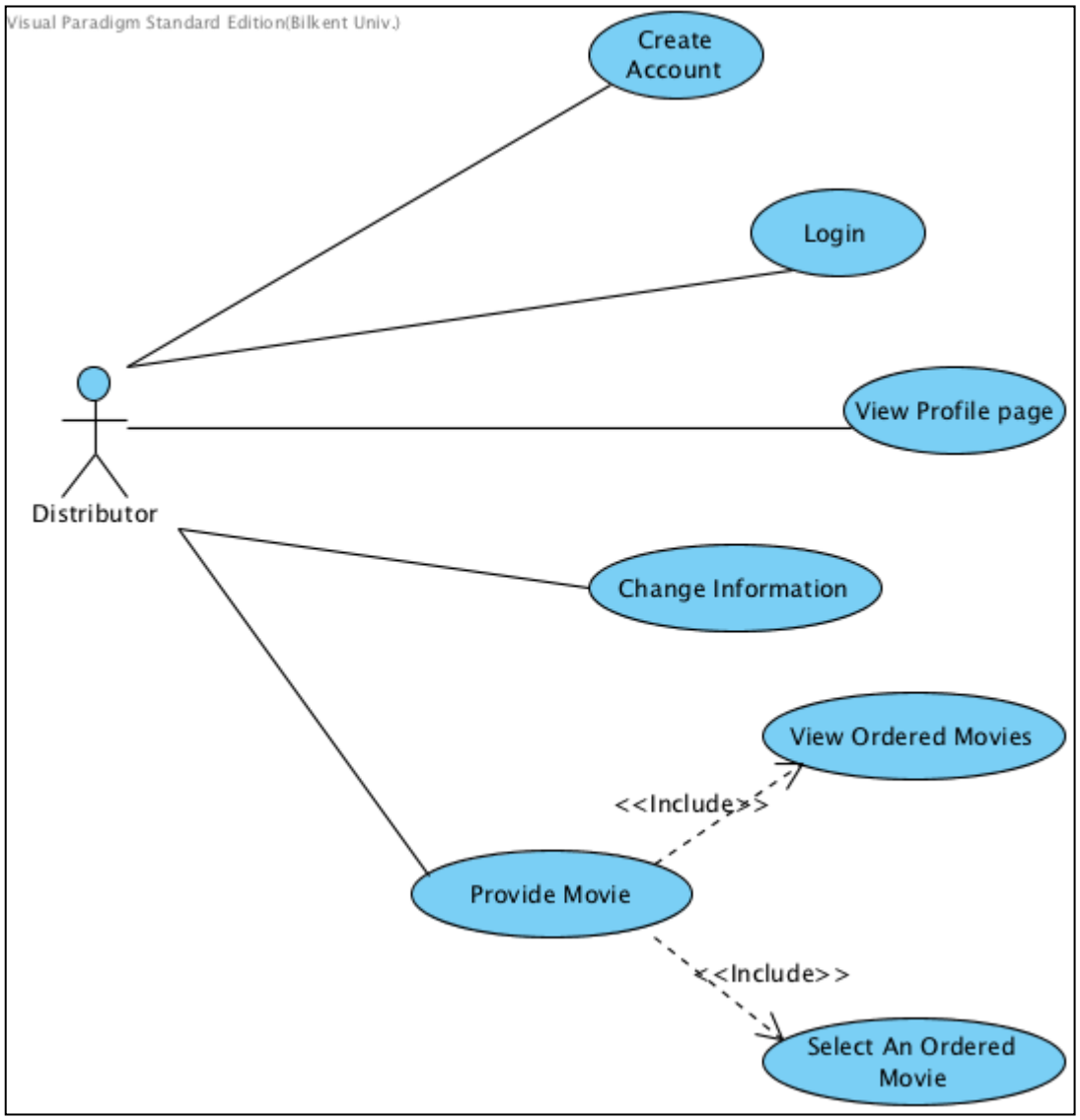

Figure 3: Distributor Use Case

5.1.4 System Admin

- System admin can login to the system using his username and password.
- System admin can view each profile on the system which includes name, age, registry number, authorization, email, address and phone.
- System admin can change his/her, and other accounts' details including password, address, and phone.
- System admin can add employee accounts to the system.
- System admin can add distributor accounts to the system.
- System admin can add a movie to the system.
- System admin can specify a maximum debt limit.
- System admin can specify a maximum rental day limit.
- System admin can add promotions to the system by specifying its details.

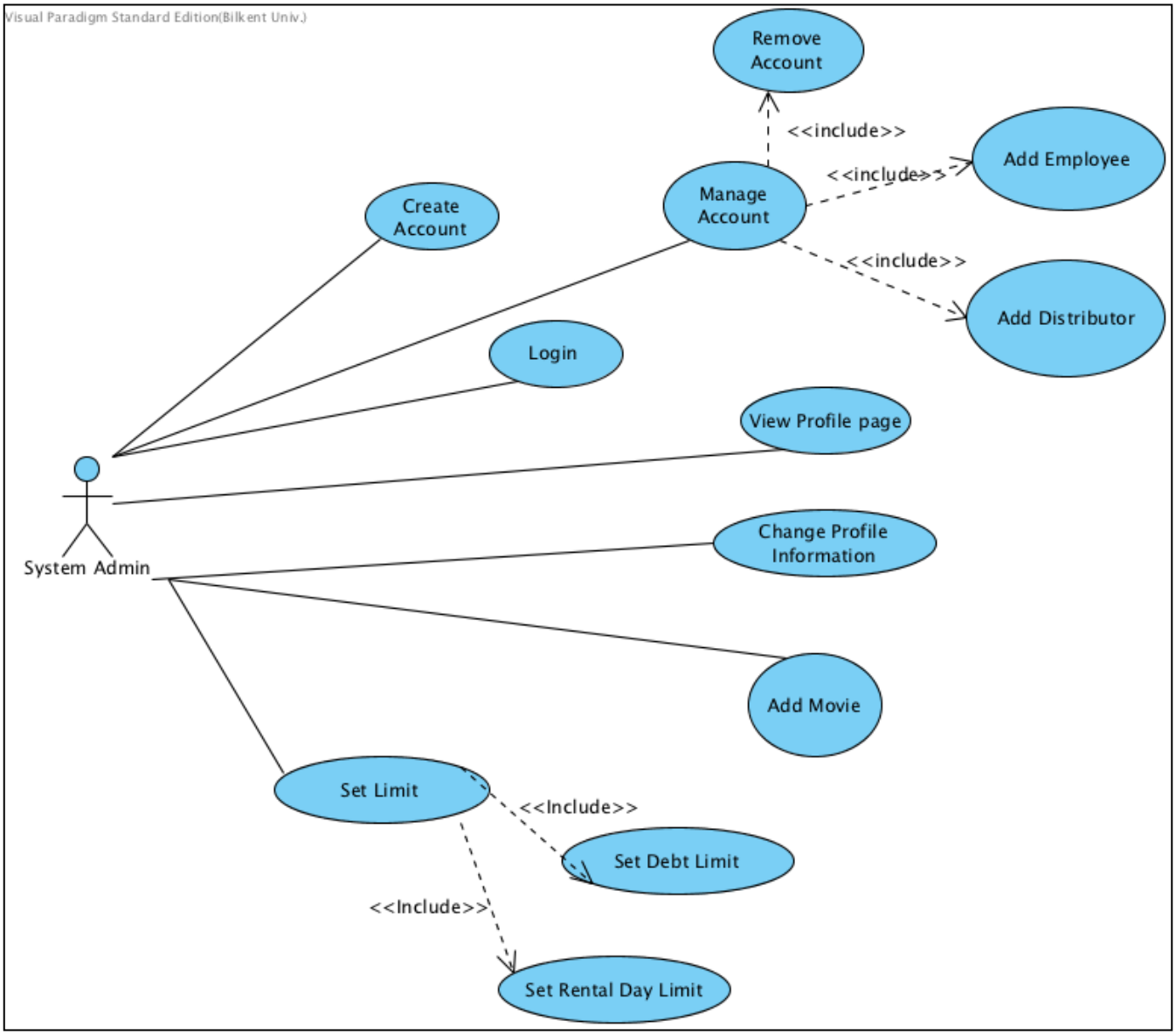

Figure 4: System Admin Use Case

5.2 Algorithms

5.2.1 Stock-Related Algorithms

In our Movie Rental Store Data Management System every movie has a stock status according to its type such as DVD or Blu-ray. It is very important to have this information about movie for buy, rent and provide tables.

By default, all the movies have a non-negative copy_number value and a non-negative rent_count value in stock table. When customer buys a movie, the copy_number of this movie in the stock table decreases by one. When customer rents a movie, the rent_count of this movie in the stock table decreases by one. When an order is provided by a distributor, the copy_number of this movie in the stock table will be increased by five.

5.2.2 Customer-Related Algorithms

5.2.2.1 Age Control

For some movies, there are some age limitations. At this point, the age attribute of the customer and age limitation attribute of the movie will be compared in order to determine whether the movie is appropriate for this user or not. If the age of the customer is greater than or equal to the age limitation of the movie, then the customer can buy or rent this movie. Otherwise the customer won't able to rent or buy this movie.

5.2.2.2 Debt control

In customer table, the current debt is kept. If the current_debt value of the customer is equal the upper limit of debt, the customer won't be able to buy or rent any movie.

5.2.2.3 Promotion Calculation

In our Movie Rental Store Data Management System, there are two different types of promotions as promotion_rent and promotion_buy. Also each promotion types have two levels. These promotions can be set to a customer when he/she satisfies several conditions. If the rent_count of the customer is equal or greater than 5, then 10% of the price_rent values of the movies will be reduced for next rent. If rent_count is equal or greater than 10, 20% of the price_rent values of the movies will be reduced for next rent. If the buy_count is equal or greater than 5, then 10% of the price_buy values of the movies will be reduced for next time

when customer buys a movie. If the buy_count is equal or greater than 10, then 20% of the price_buy values of the movies will be reduced for next time when customer buys a movie. In our system, the customer may have both promotion_rent and promotion_buy at the same time. However, the customer cannot have two promotion of same type at the same time.

5.2.3 Algorithms to Handle Logical Requirements

Since the date attributes are important for rent, they are the possible miscellaneous points in the system. In order to prevent logical errors, there should be some restrictions on them. In rent table, the rent_date attribute cannot be a future date because the system keeps only the already-made rents and buys. Also, the return_date cannot be earlier than the rent_date.

6. User Interface Design and Corresponding SQL Statements

6.1. Login

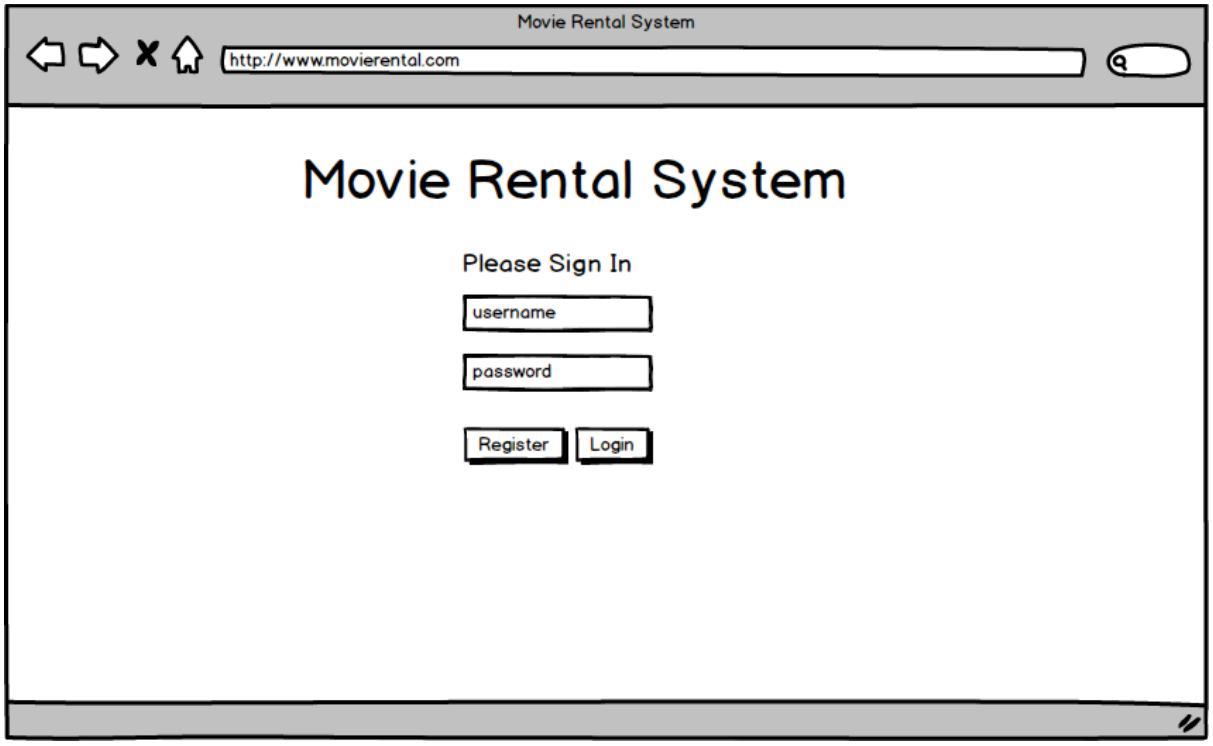

Figure 5: Login Page

Inputs: @username, @password

Process: The user enters his/her e-mail and password to login.

SQL Statements:

SELECT user_id, e_mail, password

FROM User, Distributor

WHERE (User.e_mail = @username AND User.password = @password) OR

(Distributor.e_mail = @username AND Distributor.password = @password)

6.2. Register

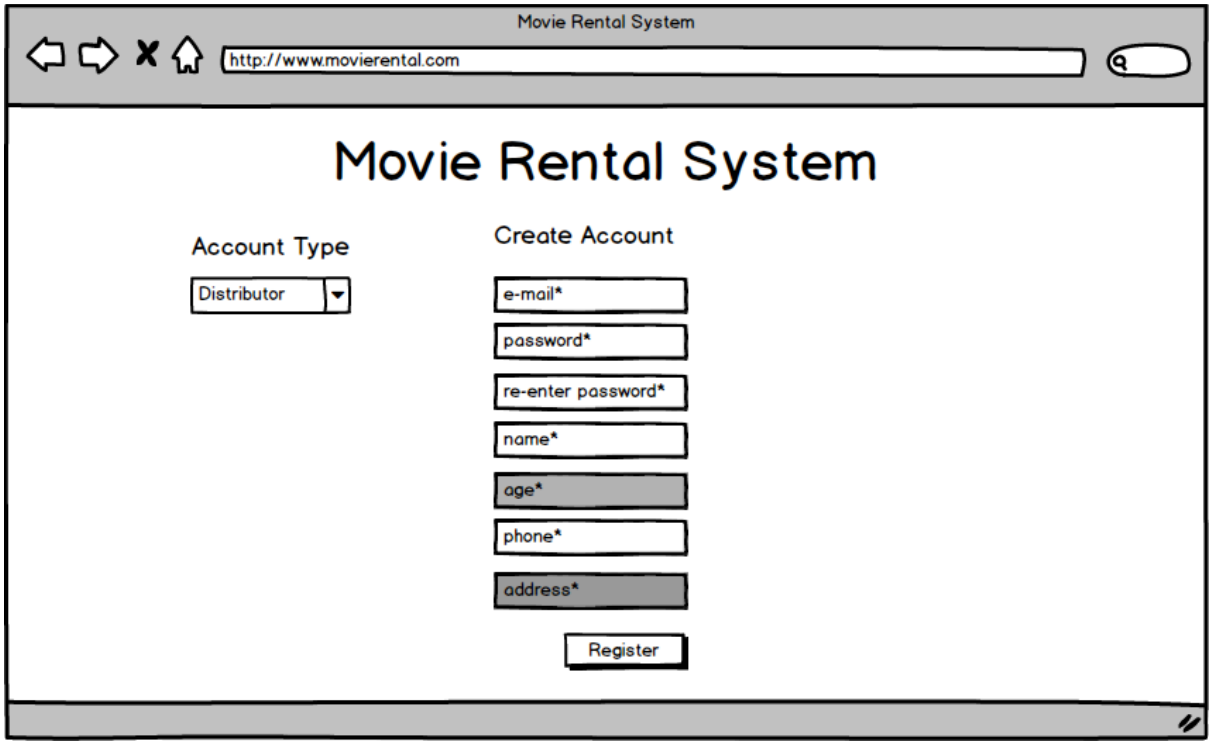

Figure 6: Register Page

Inputs: @accountType, @e-mail, @password, @name, @age, @phone, @address
Process: The user chooses his/her account type, enters required attributes depending on the account type (eg. No age attribute for distributors), then clicks register to create an account in the system.

SQL Statements:

- Account type: Distributor
  INSERT INTO Distributor VALUES (@name, @e-mail, @phone, @address, @password, 0)

- Account type: Customer
  INSERT INTO Person VALUES (@name, @age)
  INSERT INTO User VALUES ((SELECT person_id FROM Person WHERE name = @name), 0, @e-mail, @address, @phone)
  INSERT INTO Customer VALUES ((SELECT person_id FROM Person WHERE name = @name))
- Account type: Employee
  INSERT INTO Person VALUES (@name, @age)
  INSERT INTO User VALUES ((SELECT person_id FROM Person WHERE name = @name), 0, @e-mail, @address, @phone)
  INSERT INTO Employee VALUES ((SELECT person_id FROM Person WHERE name = @name))

6.3 View Customers

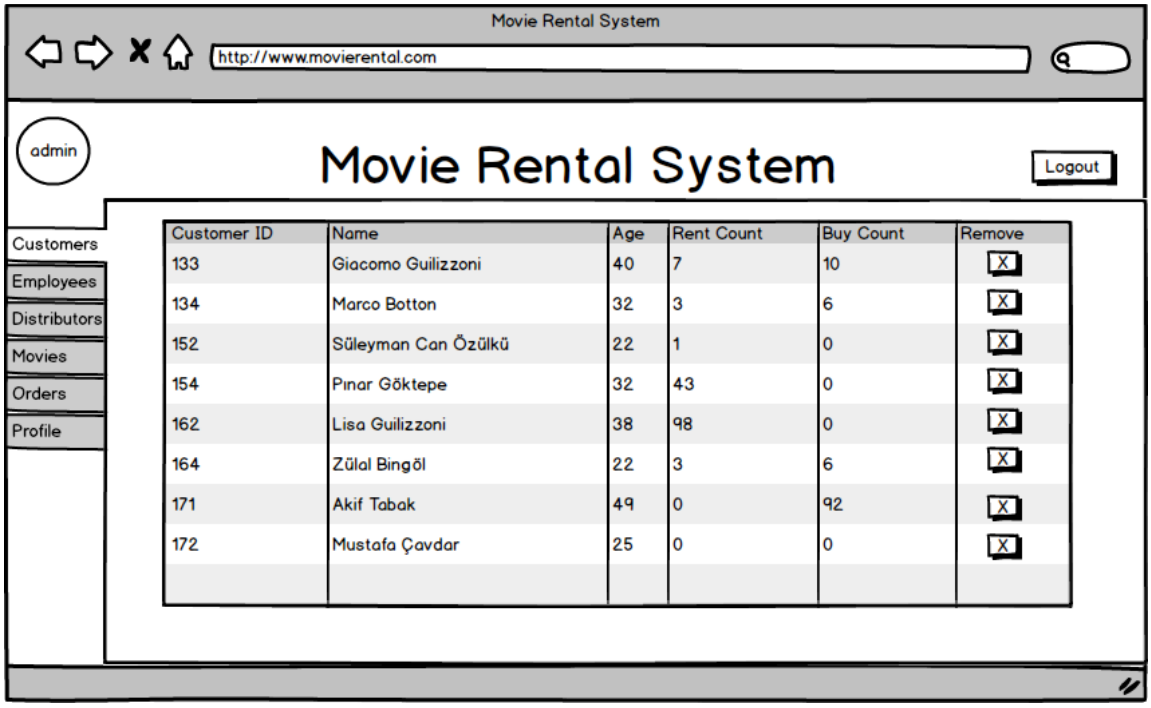

Figure 7: View Customers Page

Input: @cstmr_id (value from row where remove button is clicked)

Process: The employee with system admin authorization opens "Customers" screen which has two features: listing customers, and removing customers.

SQL Statements:

-Listing:

SELECT Customer.customer_id, Person.person_name, Person.age, Customer.rent_count, Customer.buy_count

FROM (Customer JOIN User ON (Customer.customer_id = User.user_id)) JOIN Person ON (Customer.customer_id = Person.person_id)

-Remove:

DELETE FROM Customer

WHERE customer_id = @cstmr_id

DELETE FROM User

WHERE user_id = @cstmr_id

DELETE FROM Person

WHERE person_id = @cstmr_id

6.4 Employees

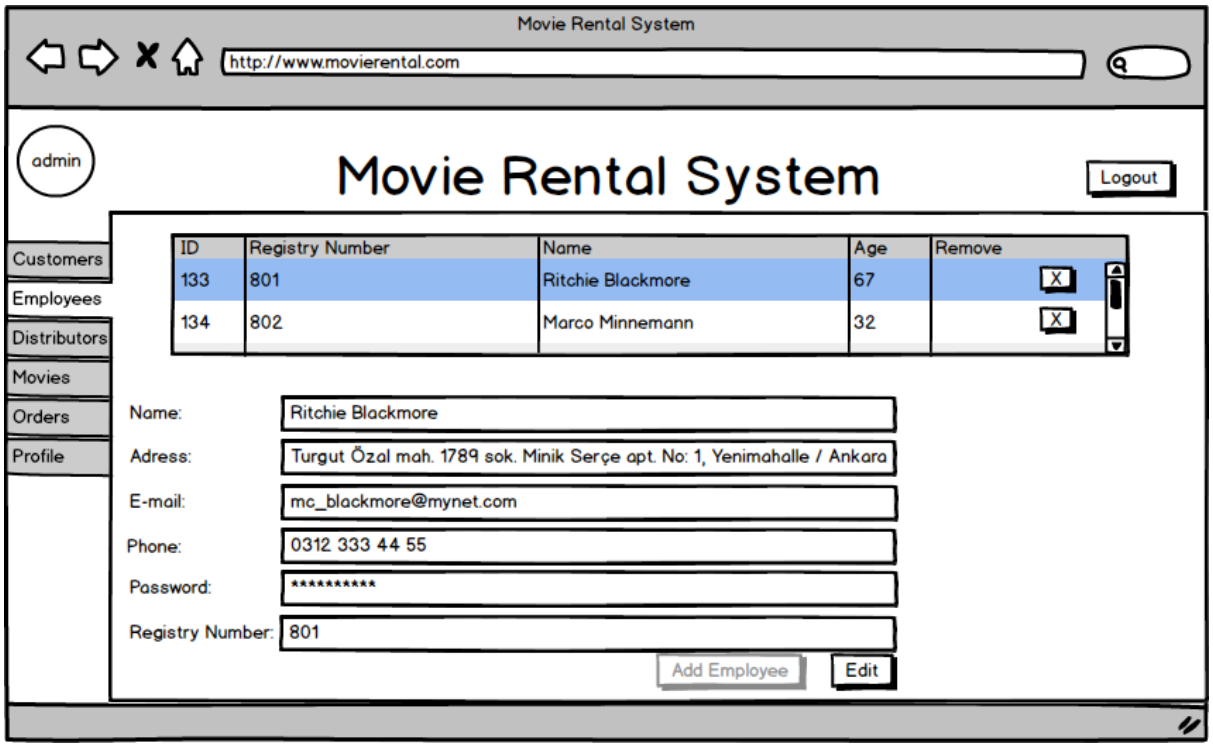

Figure 8: Employees Page

Input: @emply_id (value from row where remove button is clicked), @e-mail, @password, @name, @age, @phone, @address, @registry_number

Process: The employee with system admin authorization opens "Employees" screen which has four features: listing employees, adding employees, removing employees and editing employee information.

SQL Statements:

-Listing:

SELECT Employee.registry_number, Person.person_name, Person.age
FROM (Employee JOIN User ON(Employee.employee_id = User.user_id)) JOIN Person ON(Employee.employee_id = Person.person_id)

-Remove:

```
DELETE FROM Employee
WHERE employee_id = @emply_id
DELETE FROM User
WHERE user_id = @emply_id
DELETE FROM Person
WHERE person_id = @emply_id
```

-Adding Employee:

```
INSERT INTO Person VALUES (@name, @age)
INSERT INTO User VALUES ((SELECT person_id FROM Person WHERE name =
@name), 0, @e-mail, @address, @phone)
INSERT INTO Employee VALUES ((SELECT person_id FROM Person WHERE name =
@name))
```

-Edit Information:

```
UPDATE Person
SET person_name=@name, age=@age
WHERE person_id=@emply_id

UPDATE User
SET e_mail=@e-mail, address=@address, phone=@phone
WHERE user_id = @emply_id

UPDATE Employee
SET registry_number = @registry_number
WHERE employee_id = @emply_id
```

6.5. Distributors

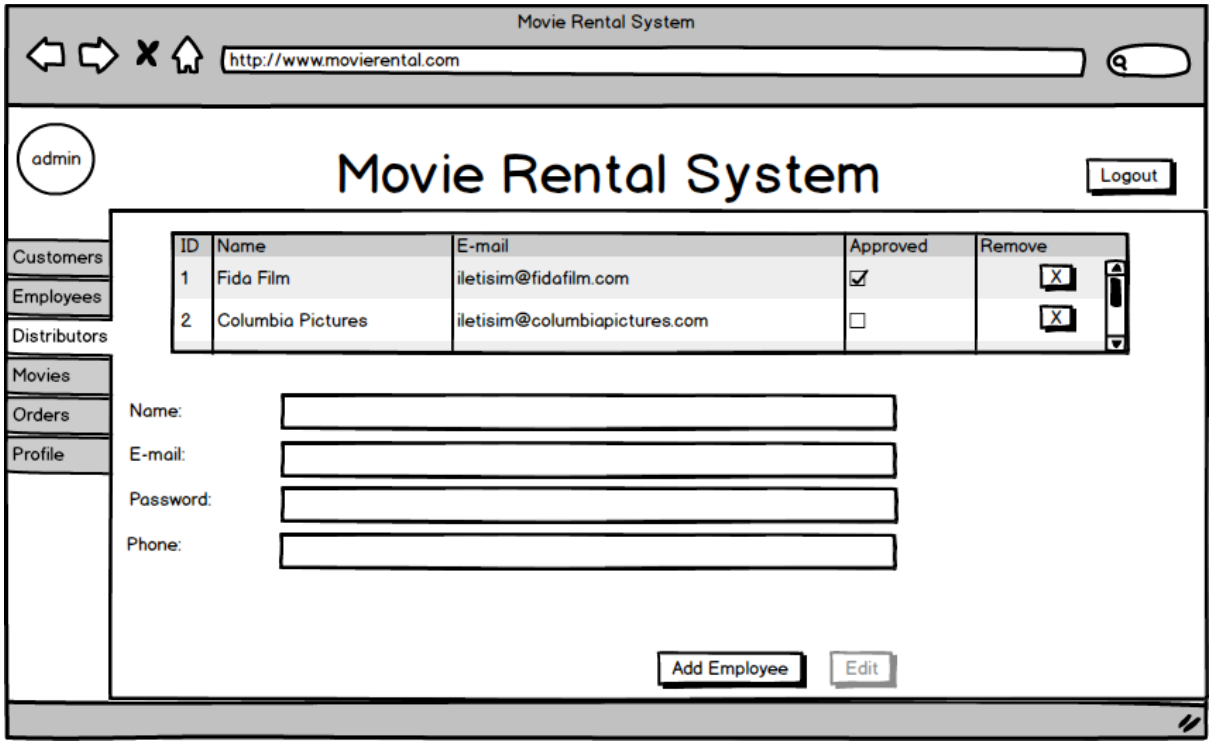

Figure 9: Distributors Page

Input: @dist_id (value from row where remove button is clicked), @e-mail, @password, @name, @phone

Process: The employee with system admin authorization opens "Distributors" screen which has five features: listing distributors, adding distributors, removing distributors and editing distributor information, approving distributor account.

SQL Statements:

-Listing:

SELECT distributor_id, distributor_name, e_mail, valid

FROM Distributor

-Approving Distributor:

UPDATE Distributor

SET valid=1

WHERE distributor_id=@dist_id

-Remove:

DELETE FROM Distributor

WHERE distributor_id = @dist_id

-Adding Distributor:

INSERT INTO Distributor VALUES (@name, @e-mail, @phone, @password)

-Edit Information:

UPDATE Person

SET distributor_name = @name, e_mail = @e-mail, phone = @phone, password = @password

WHERE distributor_id = @dist_id

6.6 Movies

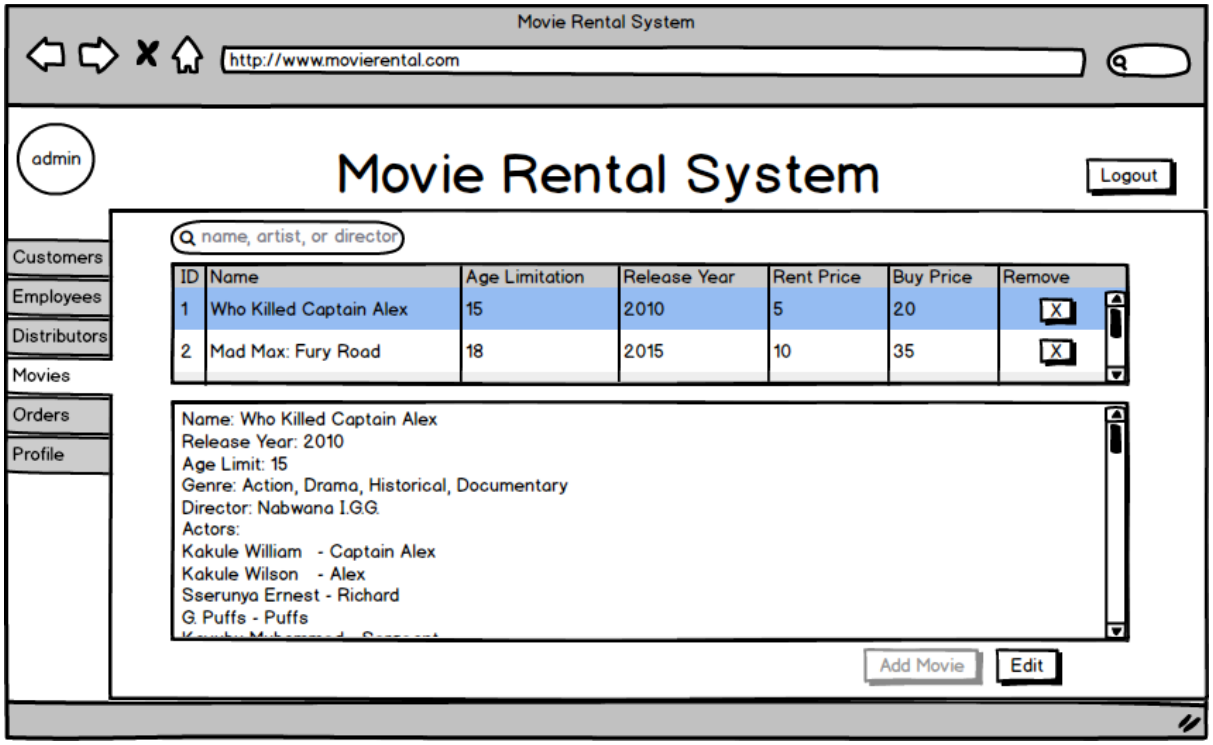

Figure 10: Movies Page

Input: @mov_id (value from row where remove button is clicked), @filter

Process: The employee with system admin authorization opens "Movies" screen which has four features: listing movies, adding movie, removing movie and editing movie information.

SQL Statements:

-Listing:

```
SELECT movie_id, movie_name, age_limitation, release_year, price_rent, price_buy,
FROM Movie
```

-Details:

```
SELECT movie_name, age_limitation, release_year, price_rent, price_buy,
FROM Movie
WHERE movie_id = @mov_id

SELECT name
FROM Movie NATURAL JOIN Play NATURAL JOIN Actor NATURAL JOIN Person
WHERE movie_id = @mov_id

SELECT name
FROM Movie NATURAL JOIN direct NATURAL JOIN Director NATURAL JOIN Person
WHERE movie_id = @mov_id
SELECT award_name, award_type, award_year
FROM Movie NATURAL JOIN Won_M NATURAL JOIN Award
WHERE movie_id = @mov_id

SELECT genre_name
FROM Movie NATURAL JOIN subject NATURAL JOIN Genre
WHERE movie_id = @mov_id
```

-Remove Movie:

DELETE FROM Movie
WHERE movie_id = @mov_id

DELETE FROM Subject
WHERE movie_id = @mov_id

DELETE FROM Won_M
WHERE movie_id = @mov_id

DELETE FROM play
WHERE movie_id = @mov_id

DELETE FROM direct
WHERE movie_id = @mov_id

-Filtering:

SELECT movie_id, movie_name, age_limitation, release_year, price_rent, price_buy,
FROM Movie NATURAL JOIN (Play NATURAL JOIN Artist) NATURAL JOIN (Direct NATURAL JOIN Director) NATURAL JOIN Person)
WHERE Person.name LIKE CONCAT ('%', @filter, '%') OR Movie.movie_name LIKE CONCAT ('%', @filter, '%')

6.7 Orders

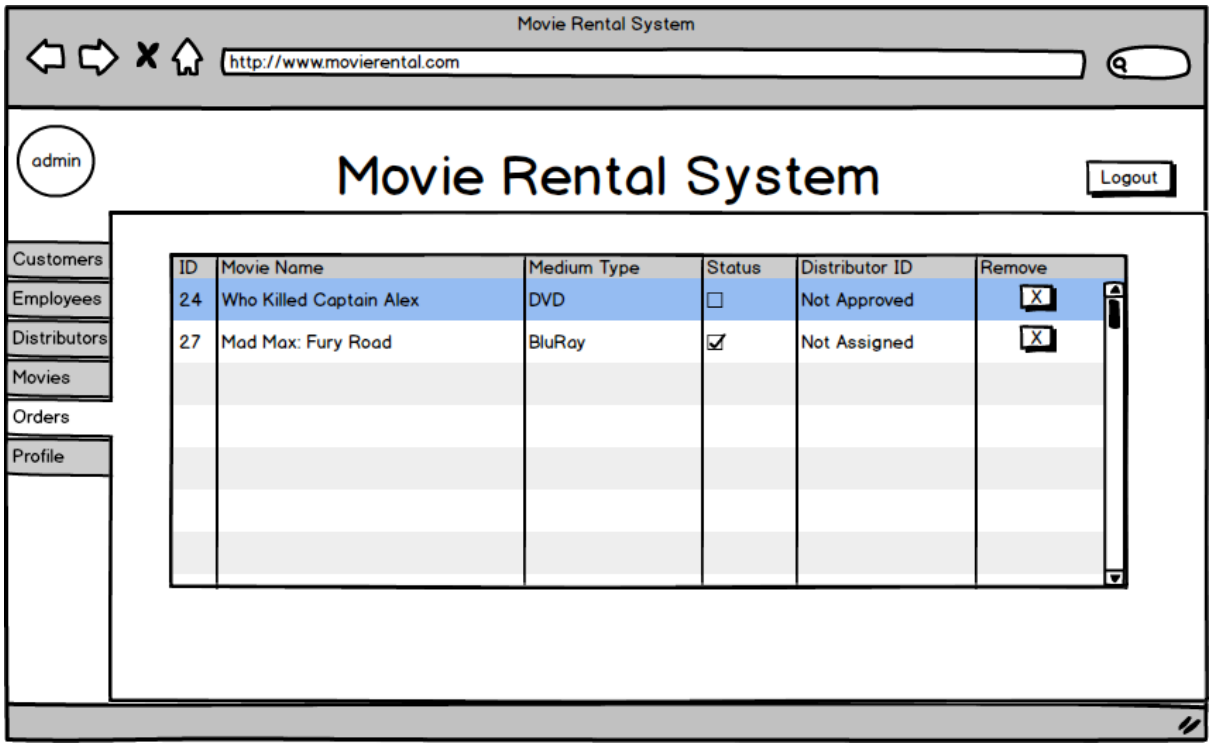

Figure 11: Orders Page

Input: @order_id (value from row where remove button is clicked or status box checked)

Process: The employee with system admin authorization opens "Orders" screen which has two features: approving order request and removing order request.

SQL Statements:

-Listing:

SELECT order_id, movie_name, movie_type, status, distributor_id

FROM OrderOfNewProducts NATURAL JOIN Order

-Remove Order:

```
DELETE FROM OrderOfNewProducts
WHERE order_id = @order_id

DELETE FROM Request
WHERE order_id = @order_id

DELETE FROM Order
WHERE order_id = @order_id
```

-Approve Order:

```
UPDATE Order
SET status=1
WHERE order_id=@order_id
```

6.8 Profile

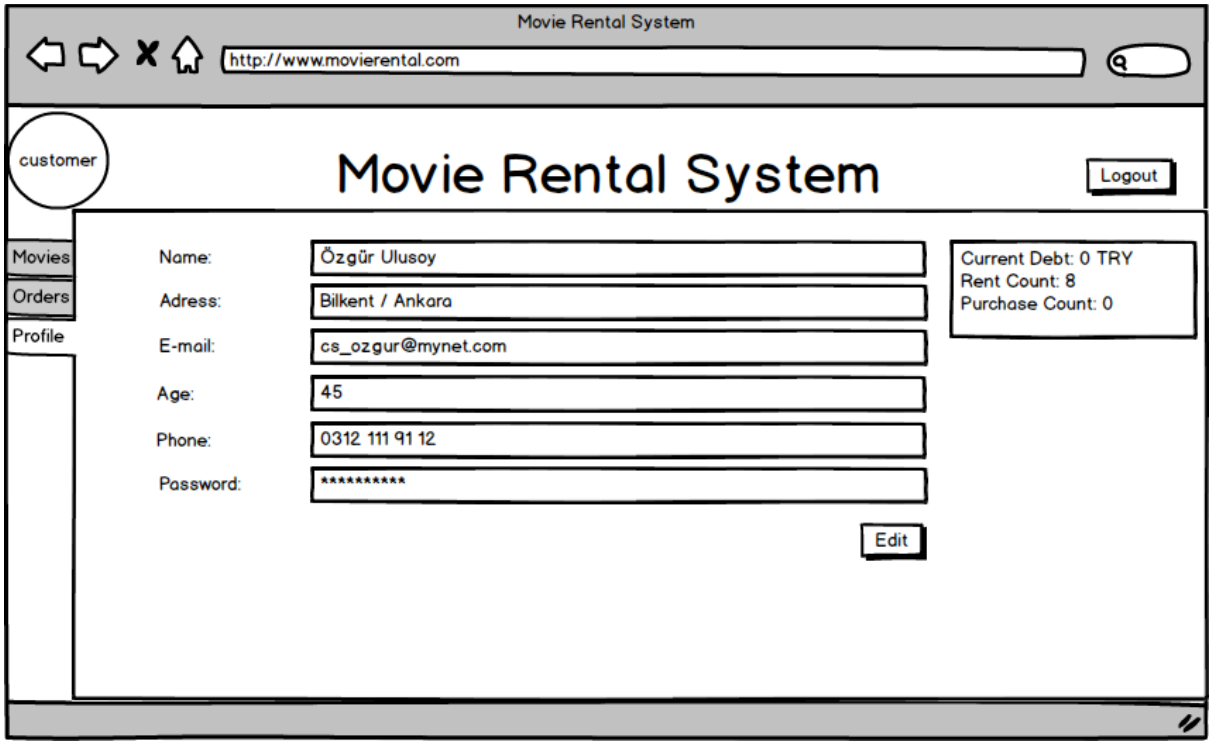

Figure 12: Profile Page

Input: @name, @address, @e-mail, @age, @phone, @password, @session_id(taken from User table at login page)

Process: The user opens "Profile" screen which has one feature: editing his/her profile.

SQL Statements:

-Editing Profile (Fields and queries may vary depending on account type of the user)**:**

```
UPDATE Person
SET person_name=@name, age=@age
WHERE person_id = @session_id
```

```
UPDATE User
SET e_mail=@e-mail, address=@address, phone=@phone, password=@
WHERE user_id = @session_id
```

-Showing Details:

```
SELECT current_debt, rent_count, buy_count
FROM Customer
WHERE customer_id = @session_id
```

6.9 Movies (Customer)

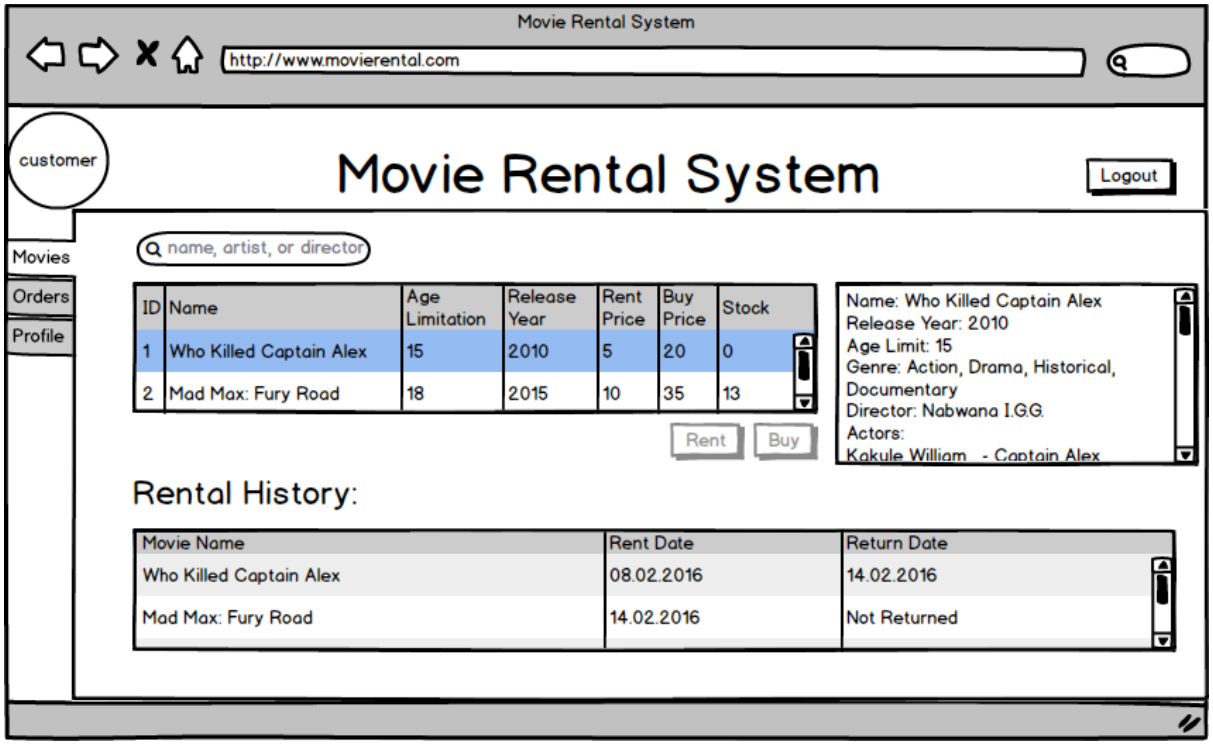

Figure 13: Movies (Customer View) Page

Input: @mov_id (value from row that is clicked), @filter, @session_id(taken from User table at login page), @date(current date formatted for SQL with PHP)

Process: The customer opens "Movies" screen which has four features: listing movies, renting movies, buying movies and listing rental history.

SQL Statements:

-Listing:

SELECT movie_id, movie_name, age_limitation, release_year, price_rent, price_buy, (copy_number - rent_count)
FROM Movie JOIN Stock ON(Movie.movie_id = Stock.stock_id)

-Details:

SELECT movie_name, age_limitation, release_year, price_rent, price_buy,
FROM Movie
WHERE movie_id = @mov_id

SELECT name
FROM Movie NATURAL JOIN Play NATURAL JOIN Actor NATURAL JOIN Person
WHERE movie_id = @mov_id

SELECT name
FROM Movie NATURAL JOIN direct NATURAL JOIN Director NATURAL JOIN Person
WHERE movie_id = @mov_id
SELECT award_name, award_type, award_year
FROM Movie NATURAL JOIN Won_M NATURAL JOIN Award
WHERE movie_id = @mov_id

SELECT genre_name
FROM Movie NATURAL JOIN subject NATURAL JOIN Genre
WHERE movie_id = @mov_id

-Filtering:

SELECT movie_id, movie_name, age_limitation, release_year, price_rent, price_buy,
FROM Movie NATURAL JOIN (Play NATURAL JOIN Artist) NATURAL JOIN (Direct NATURAL JOIN Director) NATURAL JOIN Person)
WHERE Person.name LIKE CONCAT (‘%’, @filter, ‘%’) OR Movie.movie_name LIKE CONCAT (‘%’, @filter, ‘%’)

-Renting Movie:

```
INSERT INTO Rent VALUES (@mov_id, @session_id, @date, null)

UPDATE Stock
SET rent_count = rent_count + 1
WHERE stock_id = @mov_id
```

-Buying Movie:

```
INSERT INTO Buy VALUES (@mov_id, @session_id)

UPDATE Stock
SET copy_number = copy_number - 1
WHERE stock_id = @mov_id
```

-Listing History:

```
SELECT movie_name, rent_date, return_date
FROM Rent NATURAL JOIN Movie
WHERE Rent.customer_id = @session_id
```

6.10 Orders (Customer)

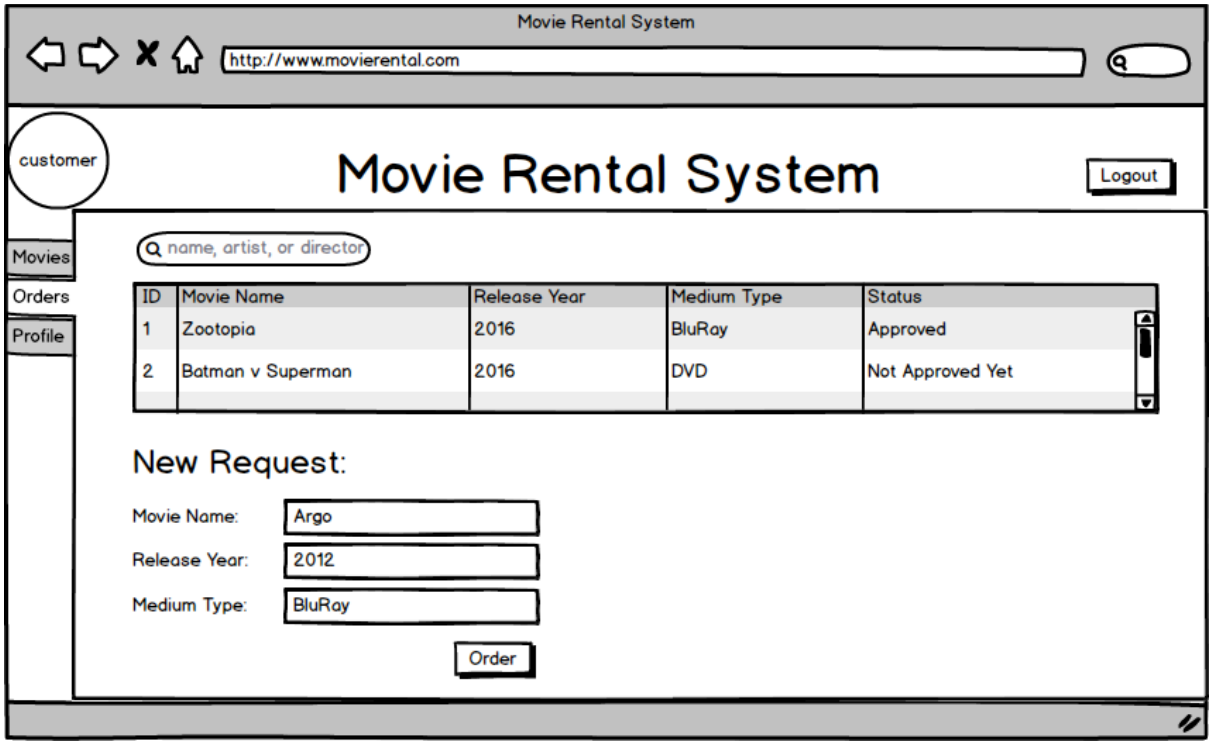

Figure 14: Orders (Customer View) Page

Input:@movie_name, @release_year, @medium_type, @session_id(taken from User table at login page)

Process: The customer opens "Orders" screen which has two features: approving order request and removing order request.

SQL Statements:

-Listing:

```
SELECT order_id, movie_name, movie_year, movie_type, status
FROM OrderOfNewProducts NATURAL JOIN Order
```

-New Request:

```
INSERT INTO OrderOfNewProducts VALUES (@movie_name, @release_year,
@medium_type)

INSERT INTO Request VALUES (@session_id, (SELECT order_id
                                          FROM OrderOfNewProducts
                                          WHERE movie_name=@movie_name AND
                                          movie_type = @medium_type))

INSERT INTO Order VALUES ((SELECT order_id
                           FROM OrderOfNewProducts
                           WHERE movie_name=@movie_name AND movie_type =
                    @medium_type), null, 0)
```

6.11 Order (Distributor)

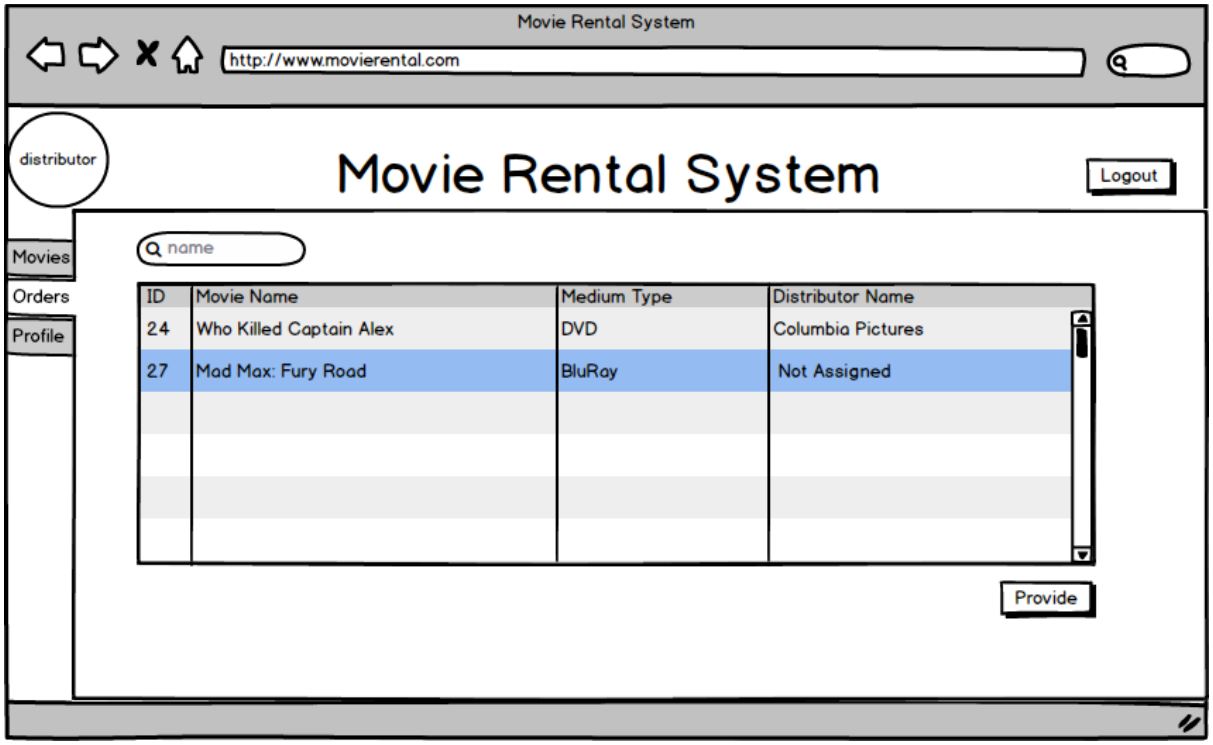

Figure 15: Orders (Distributor View) Page

Input:@order_id(value from row that is clicked), @session_id(taken from User table at login page), @filter

Process: The distributor opens "Orders" screen which has two features: listing orders, and providing order.

SQL Statements:

-Listing:

```
SELECT order_id, movie_name, movie_type, distributor_name
FROM OrderOfNewProducts NATURAL JOIN Order NATURAL JOIN Distributor
```

-Provide:

```
UPDATE Order
SET distributor_id = @session_id
WHERE order_id = @order_id
```

-Filtering

```
SELECT order_id, movie_name, movie_type, distributor_name
FROM OrderOfNewProducts NATURAL JOIN Order NATURAL JOIN Distributor
WHERE movie_name LIKE CONCAT (‘%’, @filter, ‘%’)
```

7. Advanced Database Components

7.1 Views

7.1.1 Customer Orders for Customer View

```
CREATE VIEW ordered_movies AS
    SELECT o.movie_name, a.customer_id
    FROM order o, approve a
    WHERE o.order_id = a.order_id
```

7.1.2 MovieTable for Customer View

```
CREATE VIEW movies AS
    SELECT movie_name, age_limitation, release_year, price_rent, price_buy
    FROM Movie
```

7.1.3 Rented Movies Table for Customer

```
CREATE VIEW rented_movies AS
    SELECT m.movie_name, r.rent_date, r.return_date
    FROM Rent r NATURAL JOIN Movie m
```

7.1.4 Bought Movies Table for Customer

```
CREATE VIEW bought_movies AS
    SELECT movie_name
    FROM Buy NATURAL JOIN Movie
```

7.2 Stored Procedures

- Registration process is same for all users of the system. Login process and the validity check of login information are also same for every user of the system. Therefore we can store the queries of these processes and we can execute them whenever needed.

```
CREATE PROCEDURE 'login' (@password INT, @username VARCHAR (40))
AS
BEGIN
SELECT e_mail, password
FROM User, Distributor
WHERE (user.e_mail = @username AND user.password = @password) OR
( distributor.e_mail = @username AND distributor.password = @password)
END
```

- Buy and rent procedures are the same for all customers. In both procedures, the counts of the movies are changed in the same way in stock table. So, the queries applied on these actions can be stored and executed whenever needed. In buy procedure, the copy_number is decremented whereas in rent procedure, rent_count is decremented.

7.3 Reports

7.3.1 Total Money Spent by Each Customer

SELECT customer_id, customer_name, paid_debt

FROM customer

GROUP BY customer_id

7.3.2 All Rented Movies

SELECT customer_id, customer_name, movie_id, movie_name

FROM customer c NATURAL JOIN rent r NATURAL JOIN movie m

GROUP BY customer_id

7.3.3 All Bought Movies

SELECT customer_id, customer_name, movie_id, movie_name

FROM customer c NATURAL JOIN buy b NATURAL JOIN movie m

GROUP BY customer_id

7.3.4 Waiting Order Requests

This report shows waiting order requests to employee for approval.

SELECT n.order_id, o.movie_name, o.movie_year, o.movie_type

FROM orderOfNewProducts o NATURAL JOIN order n

GROUP BY n.order_id

HAVING n.status = ‘waiting’

7.3.5 Number of Rented Movies

```
WITH customers_and_rentedMovies (customer_id, customer_name, movie_id, movie_name)
AS (SELECT c.customer_id, c.customer_name, r.movie_id, r.movie_name
        FROM customer c NATURAL JOIN rent r)
SELECT customer_id, customer_name, movie_id, movie_name, count(*) as numberOfRents
FROM customers_and_rentedMovies
GROUP BY customer_id
```

7.3.6 Number of Bought Movies

```
WITH customers_and_boughtMovies (customer_id, cutomer_name, movie_id, movie_name)
AS (SELECT c.customer_id, c.customer_name, b.movie_id, b.movie_name
        FROM customer c NATURAL JOIN buy b)
SELECT customer_id, customer_name, movie_id, movie_name, count(*) as
numberOfBuyings
FROM customers_and_boughtMovies
GROUP BY customer_id
```

7.4 Triggers

- When distributor provides the stock with movie, the stock table is updated. The "copy_number" of the corresponding movie is incremented by one.
- When a customer rents a movie from the stock,the relevant movie's stock table is updated and "rent_count" is decremented by one.
- In order to apply a promotion on the sale or rent cost, customer's "buy_count" or "rent_count" is checked. If they meet a certain amount (specified in Algorithms Section), then promotion is applied on the "price_buy" or "price_rent".
- When a customer rents a movie, the relevant customer table is updated and "rent_count" is incremented by one.
- When a customer buys a movie, the relevant customer table is updated and "buy_count" is incremented by one.

7.5 Constraints

- The system cannot be used without logging in.
- Distributor account cannot be used before system admin's approvement.
- In stock table, movies with non-positive rent_count cannot be rented.
- In stock table, movies with non-positive copy_number cannot be bought.
- If customer's age is under the age_limitation of the desired movie, this customer cannot buy or rent this movie.
- If customer's current_debt is equal to the upper limit of debt, customer cannot buy or rent any movie.
- Multiple promotions from same promotion type cannot be applied.
- Return date of the rented movie cannot be earlier than the rent date.
- Rent date cannot be future date.
- Orders, whose status is not 'submitted', cannot be added to stock.
- Promotions are only valid for single-use.

8. Implementation Plan

We will use MySQL Server as our database management system. We will use Php and JavaScript for implementing application logic and user interface.