CS353 - Database Systems

Design Report

Group 4

Online Ticket Purchasing App for Transportation

Ahmet Emir Boşnak - Section 2 - 22002398

Hasan Alp Caferoğlu - Section 1 - 22203991

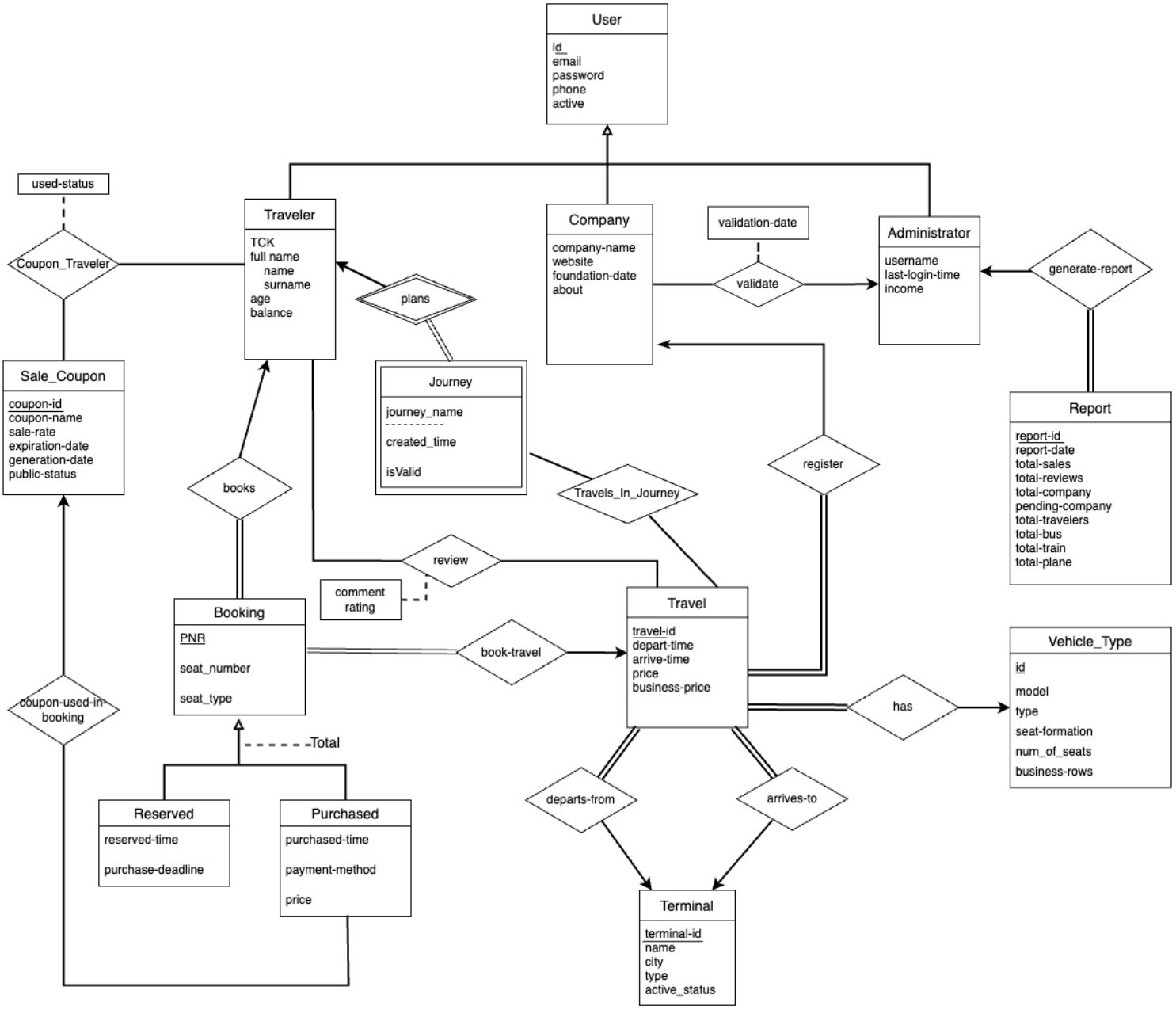
Ozan Can Gülbaz - Section 1 - 22002019

Yağız Özkarahan - Section 1 - 22002276

20 April 2023

1. Revised E/R Diagram **3**
2. Relation Schemas **5**
2.1. User 5
2.2. Traveler 6
2.3. Company 7
2.4. Administrator 8
2.5. Report 9
2.6. Sale_Coupon 10
2.7. Coupon_Traveler 11
2.8. Vehicle_Type 12
2.9. Terminal 13
2.10. Travel 14
2.11. Booking 15
2.12. Reserved 16
2.13. Purchased 17
2.14. Journey 18
2.15. Travels_In_Journey 19
2.16. Review 20
3. Functional Components **21**
3.1. Uses Cases/Scenarios 21
3.1.1. Traveler 21
3.1.2. Company 23
3.1.3. Admin 25
3.2. Algorithms 28
3.3. Data Structures 28
4. UI Design and Corresponding SQL Queries **29**
4.1. Login/Register 29
4.2. Buying a Ticket 32
4.3. Coupon 37
4.4. Balance 39
5. Advanced Database Components **40**
5.1. Views 40
5.2. Assertions 40
5.3. Triggers 41
5.4. Stored Procedures 48
6. Implementation Plan **49**

1. Revised E/R Diagram



After the proposal, the E/R diagram changed according to given feedback and possible scenarios of the application. The list of changes on the E/R diagram are as follows.

- An attribute called active is added to the User entity set. The reason for this change is that instead of deleting a user, we might deactivate that user and we can keep related information in the database. For example, after deleting a traveler, we need to take some actions such as deleting all rows related to that user or setting null for this user etc. to protect the referential integrity. However, with the help of the active attribute, admin may choose to deactivate the traveler instead of deleting.
- An attribute called activate_status is added to the Terminal entity set to check whether the terminal is in use or not. The reasons for using active attributes are valid for addition of the activate_status.
- Report entity is connected to the generate_report relationship as total participation since each report is generated by one administrator.
- In the Administrator entity set, report_date was redundant since it is already in the Report entity set. This redundant attribute is removed from the Administrator entity set.
- Register relationship had many-to-many cardinality constraints. Since each travel can be registered by only one company, the relationship cardinality constraint is changed as one-to-many with the total participation in the Travel side.
- To store the type of transportation vehicles, a new entity set called Vehicle_Type is generated which has id, model, type, seat-formation, num_of_seats and business_rows attributes. Type attribute indicates whether a vehicle is used in railroad (train) or in air (plane) or in highway (bus). The attribute seat-formation includes the information of the seating plan for a row of the vehicle. The attribute num_of_seats indicates how many seats exist in the model. And the business_rows indicated that the number of rows from the beginning that will be set used for business class passengers. Vehicle_Type entity set has a relation with Travel entity set with many-to-one cardinality constraint where total participation exists in the Travel side.
- Zip_code attribute is removed from the terminal because there are different kinds of usage for zip code and it is not necessary to store this information for our application.
- Travel entity set's attributes are changed. New attributes are travel_id, depart_time, arrive_time, price and business_price.
- In the proposal report, there was a seat weak entity set. This weak entity set is changed to a Booking entity set which is a strong entity set. It has attributes which are PNR, seat_number and seat_type. A specialization is applied for the Booking entity set. Reserved and Purchased entity sets are generated and they are connected to the Booking entity set as disjoint specialization. Since these two entity sets are generated, reserved and purchased relationships which were in the old E/R model are removed. Instead of these two relationships, one relationship between Booking and Traveler entity sets are used which is named as book.

- Purchased entity set in the ER diagram has purchased_time, payment_method and price attributes. Price attribute is stored for a payment because a price might be updated for a travel and we want to store the price which a traveler pay.
- A new attribute called public_status is added to the Sale_Coupon entity set. The purpose of this attribute is to store whether the generated coupon is available from all users or not.
- A relationship called coupon_used_in_booking is added to the E/R diagram. The reason is to store whether a coupon is used in a purchase or not and store which coupon is used if any coupon is used.
- Additionally, a journey entity set is added to the E/R diagram. Journey entity set has journey_name, created_time and isValid attributes. This entity set has relation with both Traveler and Travel entity sets since a journey describes a list of several travels for a traveler. Since each journey is generated by the user and they are user specific, Journey is a weak entity set whose owner entity set is Traveler. Without travelers, there wouldn't be any journey. Travels_In_Journey has many-to-many relationships with Journey and Travel entity sets.

2. Relation Schemas

2.1. User

Table Schema: User(id, email, password, phone, active)

Candidate Keys: id, email, phone

Primary Key: id

Functional Dependencies: F = {
id → email, password, phone, active,
email → id,
phone → id }

Normal Form: Attribute email functionally determines the id (email → id) and also attribute phone functionally determines the id (phone → id). Since attribute id is selected as primary key which means functionally determines all attributes, email and phone are superkey for the User relation. So, the relation satisfies the condition of BCNF.

Table Creation:

```
CREATE TABLE User(
      id INT AUTO_INCREMENT,
      email VARCHAR(256) NOT NULL,
      password VARCHAR(256) NOT NULL,
      phone VARCHAR(256) NOT NULL,
      active BOOLEAN NOT NULL DEFAULT TRUE,
      PRIMARY KEY (id),
      UNIQUE (email),
      UNIQUE (phone)
);
```

2.2. Traveler

Table Schema: `Traveler(id, TCK, name, surname, age, balance)`

Candidate Keys: { id, TCK }

Primary Key: { id }

Functional Dependencies: F = {
id → TCK, name, surname, age, balance,
TCK → id }

Normal Form: Attribute TCK functionally determines the id (TCK → id) Since attribute id is the primary key which means functionally determines all attributes, TCK is a super key for the Traveler relation. So, the relation satisfies the BCNF conditions. TCK is a candidate key because TCK is unique (superkey) and minimal.

Table Creation:

```
CREATE TABLE Traveller(
      id INT,
      TCK VARCHAR(50) NOT NULL,
      name VARCHAR(50) NOT NULL,
      surname VARCHAR(50) NOT NULL,
      age TINYINT NOT NULL,
      balance DECIMAL(10,2) DEFAULT 0,
      PRIMARY KEY(id),
      FOREIGN KEY(id) REFERENCES to User
            ON DELETE CASCADE,
      UNIQUE (TCK),
      CONSTRAINT check_age,
      CHECK ( age > 18 )
);
```

2.3. Company

Table Schema: `Company(id, company_name, website, foundation_date, about, validator_id, validation_time)`

Candidate Keys: { id, company_name, website }

Primary Key: id

Functional Dependencies: F = {
id → company_name, website, foundation_date, about, validator_id, validation_time,
company_name → id,
website → id }

Normal Form: Attribute company_name functionally determines the id and website functionally determines the id. Since attribute id is the primary key which functionally determines all attributes, company_name and website are super keys. So, the relation satisfies the BCNF conditions. company_name and website are also candidate keys because both attributes are unique (superkey) and minimal.

Table Creation:

```
CREATE TABLE Company(
      id INT,
      company_name VARCHAR(256),
      website VARCHAR(256),
      foundation_date DATE,
      about TEXT DEFAULT NULL,
      validator_id INT DEFAULT NULL,
      validation_date DATETIME DEFAULT NULL,
      PRIMARY KEY(id),
      FOREIGN KEY(id) REFERENCES to User
            ON DELETE CASCADE,
      FOREIGN KEY(validator_id) REFERENCES Administrator(id)
            ON DELETE SET DEFAULT,
      UNIQUE (company_name),
      UNIQUE (website)
);
```

2.4. Administrator

Table Schema: `Administrator(id, username, last_login_time, income)`

Candidate Keys: { id, username }

Primary Key: id

Functional Dependencies: F = {
id → username, last_login_time, income,
username → id }

Normal Form: Attribute username functionally determines the id (username → id) Since attribute id is the primary key which means functionally determines all attributes, username is a super key. So, the relation satisfies the BCNF conditions. Attribute username is a candidate key because it is unique (superkey) and minimal.

Table Creation:

```
CREATE TABLE Administrator(
      id INT,
      username VARCHAR(256) NOT NULL,
      last_login_time DATETIME DEFAULT NULL,
      income DECIMAL(10,2) NOT NULL DEFAULT 0.00
      PRIMARY KEY(id),
      FOREIGN KEY(id) references to User,
            ON DELETE CASCADE
      UNIQUE(username)
);
```

2.5. Report

Table Schema: Report(report_id, report_date, total_sales, total_reviews, total_travels, total_company, pending_company, total_travelers, total_bus, total_train, total_plane, report_generator_id)

Candidate Keys: report_id, report_date

Primary Key: report_id

Functional Dependencies: F = {
report_id → report_date, total_sales, total_reviews, total_travels, total_company, pending_company, total_travelers, total_bus, total_train, total_plane, report_generator_id,
report_date → report_id }

Normal Form: The functional dependency report_date → report_id satisfies the BCNF conditions.

Table Creation:

```
CREATE TABLE Report(
      report_id INT,
      report_date DATETIME,
      total_sales DECIMAL(15,2),
      total_reviews INT,
      total_company INT,
      pending_compan INT,
      total_travelers INT,
      total_bus INT,
      total_train INT,
      total_plane INT,
      report_generator_id INT,
      PRIMARY KEY(report_id),
      FOREIGN KEY(report_generator_id) REFERENCES Administrator(id)
            ON DELETE SET NULL,
      UNIQUE (report_date)
);
```

2.6. Sale_Coupon

Table Schema: `Sale_Coupon(coupon_id, coupon_name, sale_rate, expiration_date, generation_date, public_status)`

Candidate Keys: coupon_id

Primary Key: coupon_id

Functional Dependencies: F = {
coupon_id → coupon_name, sale_rate, expration_date, generation_date, puclic_status }

Normal Form: Attribute coupon_id functionally determines all attributes of the relation. For that reason it is super key. This indicates that the relation is in BCNF. Since coupon_id is unique and minimal, coupon_id is a candidate key. The only candidate key is coupon_id and it is selected as the primary key.

Table Creation:

```
CREATE TABLE Sale_Coupon(
      coupon_id INT,
      coupon_name VARCHAR(20) NOT NULL,
      sale-rate DECIMAL(3,2) NOT NULL,
      expiration_date DATE NOT NULL,
      generation_date DATE NOT NULL,
      public-status ENUM( 'public', 'private' ) NOT NULL,
      PRIMARY KEY(coupon_id),
      CONSTRAINT rate_check
      CHECK (sale_rate < 1),
      CONSTRAINT check-same-active-coupon
      CHECK ( NOT EXISTS ( SELECT *
                           FROM Sale_Coupon SC1, Sale_Coupon SC2
                           WHERE SC1 <> SC2
                               AND SC1.coupon_name = SC2.coupon_name
                               AND SC2.generation_date < SC1.expiration_date)
);
```

2.7. Coupon_Traveler

Table Schema: `Coupon_Traveler(coupon_id, user_id, used_status)`

Candidate Keys: (coupon_id, user_id)

Primary Key: (coupon_id, user_id)

Functional Dependencies: F = { coupon_id, user_id → used_status}

Normal Form: Both coupon_id and user_id functionally determine the used_status. This set of attributes (coupon_id, user_id) is super key for the relation. For that reason, the relation is in BCNF. Additionally this set of attributes is minimal which makes it only candidate key.

Table Creation:

```
CREATE TABLE Coupon_Traveler(
      coupon_id INT,
      user_id INT,
      used_status BOOLEAN DEFAULT FALSE,
      PRIMARY KEY(coupon_id, user_id),
      FOREIGN KEY(coupon_id) REFERENCES Sale_Coupons(coupon_id),
            ON DELETE CASCADE,
      FOREIGN KEY(user_id) REFERENCES Traveler(id)
            ON DELETE CASCADE
);
```

2.8. Vehicle_Type

Table Schema: `Vehicle_Type(id, model, type, seat_formation, num_of_seats, business_rows)`

Candidate Keys: `id, model`

Primary Key: `id`

Functional Dependencies: F = {
id → model, type, seat_formation, num_of_seats, business_rows
model → id }

Normal Form: Attribute model functionally determines the id (model → id) Since attribute id is the primary key which means functionally determines all attributes, model is a super key. So, the relation satisfies the BCNF conditions. Attribute model is also a candidate key because it is unique (superkey) and minimal.

Table Creation:

```
CREATE TABLE Vehicle_Type(
      id INT AUTO_INCREMENT,
      model VARCHAR(256) NOT NULL,
      type VARCHAR(256) NOT NULL,
      seat_formation VARCHAR(256) NOT NULL,
      num_of_seats INT NOT NULL DEFAULT 0,
      business_rows INT NOR NULL DEFAULT 0,
      PRIMARY KEY (id),
      UNIQUE ( model )
);
```

2.9. Terminal

Table Schema: `Terminal(terminal_id, name, city, type, activate_status)`

Candidate Keys: `terminal_id, name`

Primary Key: `terminal_id`

Functional Dependencies: F = {
terminal_id → name, city, type, activate_status
name → terminal_id }

Normal Form: Attributes name functionally determines the terminal_id. Since terminal_id is the primary key which means functionally determines all attributes, name is a super key. So, the relation satisfies the BCNF conditions. Attributes name is also a candidate key because it is unique (superkey) and minimal.

Table Creation:

```
CREATE TABLE Terminal(
      terminal_id INT AUTO_INCREMENT,
      name VARCHAR(256) NOT NULL,
      city VARCHAR(256) NOT NULL,
      type ENUM( 'plane', 'train', 'bus' ) NOT NULL,
      active_status ENUM( ‘active’, ‘inactive’),
      PRIMARY KEY (terminal_id),
      UNIQUE (name)
);
```

2.10. Travel

Table Schema: `Travel(travel_id, travel_company_id, departure_terminal_id, arrival_terminal_id, depart_time, arrive_time, price, business-price, vehicle_type_id)`

Candidate Keys: travel_id

Primary Key: travel_id

Functional Dependencies: F = {
travel_id → travel_company_id, departure_terminal_id, arrival_terminal_id, depart_time, travel_duration, arrive_time, price, business-price, vehicle_type_id
}

Normal Form: Attribute travel_id is super key which functionally determines all the attributes of the relation. Attribute travel_id is also candidate key since it is unique and minimal. It is also selected as the primary key.

Table Creation:

```
CREATE TABLE Travel(
      travel_id INT AUTO_INCREMENT,
      travel_company_id INT NOT NULL,
      departure_terminal_id INT NOT NULL,
      arrival_terminal_id INT NOT NULL,
      depart_time DATETIME NOT NULL,
      arrive_time DATETIME NOT NULL,
      price  DECIMAL(10,2) NOT NULL,
      business_price DECIMAL(10,2),
      vehicle_type_id INT,
      PRIMARY KEY(travel_id),
      FOREIGN KEY(travel_company_id) REFERENCES Company(id)
            ON DELETE CASCADE,
      FOREIGN KEY(departure_terminal_id) REFERENCES Terminal(id)
            ON DELETE SET NULL,
      FOREIGN KEY(arrival_terminal_id) REFERENCES Terminal(id)
            ON DELETE SET NULL,
      FOREIGN KEY(vehicle_type_id) REFERENCES Vehicle_Type(id)
            ON DELETE SET NULL,
      CHECK (arrive_time > depart_time));
```

2.11. Booking

Table Schema: Booking(PNR, travel_id, seat_number, seat_type traveler_id)

Candidate Keys: PNR

Primary Key: PNR

Functional Dependencies: F = { PNR → travel_id, seat_number, seat_type, traveler_id }

Normal Form: Attribute PNR functionally determines all the attributes of the relation, so it is a super key. PNRis also a candidate key since it is unique and minimal. It is also selected as the primary key.

Table Creation:

```
CREATE TABLE Booking(
      PNR VARCHAR(20),
      travel_id INT,
      seat_number INT N0T NULL,
      traveler_id INT,
      seat-type ENUM( “business”, “regular” ) NOT NULL
            DEFAULT “regular”,
      PRIMARY KEY(PNR),
      FOREIGN KEY (travel_id) REFERENCES Travel
            ON DELETE CASCADE,
      FOREIGN KEY (traveler_id) REFERENCES Traveler
            ON DELETE CASCADE
);
```

2.12. Reserved

Table Schema: `Reserved(PNR, reserved_time, purchase_deadline)`

Candidate Keys: PNR

Primary Key: PNR

Functional Dependencies: F = { PNR → reserved_time, purchase_deadline }

Normal Form: Attribute PNR functionally determines all the attributes of the relation, so it is a super key. PNR is also a candidate key since it is unique and minimal. It is also selected as the primary key.

Table Creation:

```
CREATE TABLE Reserved(
      PNR VARCHAR(20),
      reserved_time DATETIME NOT NULL,
      purchased_deadline DATETIME NOT NULL,
      PRIMARY KEY(PNR),
      FOREIGN KEY (PNR) REFERENCES Booking
            ON DELETE CASCADE
);
```

2.13. Purchased

Table Schema: `Purchased(PNR, purchased_time, payment_method, price, coupon_id)`

Candidate Keys: PNR

Primary Key: PNR

Functional Dependencies: F = { PNR → purchased_time, payment_method, price, coupon_id)

Normal Form: Attribute PNR functionally determines all the attributes of the relation, so it is a super key. PNRis also a candidate key since it is unique and minimal. It is also selected as the primary key.

Table Creation:

```
CREATE TABLE Reserved(
      PNR VARCHAR(20),
      purchased_time DATETIME NOT NULL,
      payment_method VARCHAR(20) NOT NULL,
      price INT NOT NULL DEFAULT 0,
      coupon_id INT DEFAULT NULL,
      PRIMARY KEY(PNR),
      FOREIGN KEY (PNR) REFERENCES Booking
            ON DELETE CASCADE,
      FOREIGN KEY (coupon_id) REFERENCES Sale_Coupon
            ON DELETE SET NULL
);
```

```
NOTE: Price attributes stores the price of the ticket when the
ticket is purchased. The price may change for the travel after a
traveler purchase a ticket.
```

2.14. Journey

Table Schema: `Journey(journey_name, traveler_id, created_time, isValid)`

Candidate Keys: {journey_name, traveler_id}

Primary Key: { journey_name, traveler_id }

Functional Dependencies: F = { journey_name, traveler_id → created_time, isValid }

Normal Form: The set of attributes journey_name and traveler_id functionally determine all the attributes of the relation, so this set of attributes is a super key. It is also a candidate key since it is unique and minimal. It is also selected as the primary key.

Table Creation:

```
CREATE TABLE Journey(
      journey_name VARCHAR(256) NOT NULL,
      traveler_id INT,
      created_time DATETIME NOT NULL,
      isValid ENUM('valid', 'not valid') NOT NULL DEFAULT 'valid',
      PRIMARY KEY (journey_name, traveler_id),
      FOREIGN KEY (traveler_id) REFERENCES Traveler,
            ON DELETE CASCADE
);
```

2.15. Travels_In_Journey

Table Schema: `Travels_In_Journey(journey_name, traveler_id, travel_id)`

Candidate Keys: { journey_name, traveler_id, travel_id }

Primary Key: { journey_name, traveler_id, travel_id }

Functional Dependencies: F = {
journey_name, traveler_id, travel_id → journey_name, traveler_id, travel_id)

Normal Form: Functional dependency written above is trivial. The relation is in BCNF.

Table Creation:

```
CREATE TABLE Travels_In_Journey(
      journey_name VARCHAR(256) NOT NULL,
      traveler_id INT,
      travel_id INT,
      PRIMARY KEY (journey_id, traveler_id, travel_id),
      FOREIGN KEY (journey_name) REFERENCES Journey,
            ON DELETE CASCADE
      FOREIGN KEY (traveler_id) REFERENCES Traveler,
            ON DELETE CASCADE
      FOREIGN KEY (travel_id) REFERENCES Travel
            ON DELETE CASCADE
);
```

2.16. Review

Table Schema: `Review(travel_id, traveler_id, comment, rating)`

Candidate Keys: { travel_id, traveler_id }

Primary Key: { travel_id, traveler_id }

Functional Dependencies: F = {
travel_id, traveler_id → comment, rating)

Normal Form: The set of attributes travel_id and traveler_id functionally determine all the attributes of the relation, so this set of attributes is a super key. It is also a candidate key since it is unique and minimal. It is also selected as the primary key.

Table Creation:

```
CREATE TABLE Reserved(
      travel_id INT,
      traveler_id INT,
      comment TEXT DEFAULT NULL,
      rating INT NOT NULL CHECK (rating >= 1 AND rating <= 5),
      travel_id INT,
      PRIMARY KEY (travel_id, traveler_id),
      FOREIGN KEY (travel_id) REFERENCES Travel,
            ON DELETE SET NULL
      FOREIGN KEY (traveler_id) REFERENCES Traveler
            ON DELETE CASCADE
);
```

3. Functional Components

3.1. Uses Cases/Scenarios

3.1.1. Traveler

1) Creating account:
    a) User fills out the blanks for email, password, phone, TCK, name, surname and age. Then press register button
        i) The system check whether there is an existing user with the same email
    b) User profile is created

2) Login to the account:
    a) User fills out the blank for email and password.
        i) The system checks if there is an existing user with this email
        ii) If there is a user, then the system checks whether the password is correct or not.
    b) Show main page

3) View / Change Traveler Profile:
    a) The system only shows travelers' their own profile.
    b) The user clicks on the edit profile button
    c) Enter new values for one or several of the new password, phone, name, surname, age.
        i) The system finds the user in the database and updates corresponding values.

4) Adding money to balance:
    a) Traveler enters card information and amount of money to deposit his/her balance.
        i) The system checks the existence of the user and finds the user
        ii) The system updates the balance.

5) View / Add Coupon:
    a) Travelers go to the page which shows their coupons.
    b) Traveler tries to add coupon by entering the name of the coupon
        i) The system checks if there is an available coupon with this name by looking for expiration date
        ii) The system checks coupon whether public or private
        iii) If the coupon is public and hasn't been added by this user yet, add this coupon to the Coupon_Traveler relation.
        iv) If the coupon is private, check Coupon_Traveler relation whether this coupon is added by another user. If this coupon has already been added by another user, then don't add this coupon for the traveler. If this coupon has not been added by any traveler until now, then add this coupon to the traveler who enters this coupon name.
    c) System shows previously used and available coupons. Coupons might be public or private to the user. Available coupons can be used by the user during purchasing tickets.

6) View all travels:
    a) Users can see all their past and upcoming travels.
    b) User can filter according to date of the travel

7) Buying a ticket:
    a) Apply filters if necessary (e.g., departure, destination, transport type, etc.)
    b) List all possible companies and/or timetables
    c) The user selects the vehicle and date/time.
        i) Trip restrictions are checked.
        ii) The system checks if the trip's date/time collides with any trips on the user's travel list.
    d) Show / Enter the user's payment method.
    e) User selects a coupon among the available coupons if it is wanted to be used.
    f) Make the payment:
        i) The system checks the expiration time of the selected coupon if any coupon is selected.
        ii) The system checks whether there is enough money in the user's balance.
        iii) The money is deducted from the balance.
    g) The ticket is added to the user's (upcoming) travel list.

8) Planning a journey:
    a) Enter a name for the journey (i.e., summer holiday)
    b) List all possible companies and / or timetables
    c) Apply filters if necessary (e.g., departure, destination, transport type, etc.)
    d) Select multiple trips and add them to the current journey.
    e) The journey is stored in the user account.
        i) "reserve all" or "buy all tickets" options are provided for the user.
            (1) To reserve all tickets for the journey, users will be directed to select a seat for each travel.

9) Rating and adding a comment to a travel of a company:
    a) A traveler can comment on their past travels.
        i) The system lists all past travels of the traveler.
        ii) Click on the comment button for one of the listed travels.
        iii) Enter rating and write comment

10) Editing previously added comment
    a) A traveler can only edit his/her previously written comment
        i) A traveler enters a rating and comments.

3.1.2. Company

1) List the company's all travels:
    a) A company can view all their past and upcoming travels.

2) Register a travel:
    a) A company enters departure terminal, arrival terminal, departure time, arrival time, business price and vehicle type
    b) Publish the travel

3) View details of travel:
    a) A company select one of their travels
    b) View all reservations and purchases related to the selected travel.
    c) If the selected travel has completed, view comments related to this travel.

4) Delete a travel:
    a) A company can delete only their upcoming travels
        i) The systems check if there is such travel and check if the travel has completed or not.
        ii) If the travel which is desired to be deleted is upcoming travel, then delete all reservations and purchases on the travel. It is the company's responsibility to make new bookings for the travelers who have a reservation or have purchased a ticket.

5) Update a travel:
    a) A company can update a travel detail by entering departure time, arrival time, price, etc.
        i) The system checks if the travel is completed or upcoming. A company cannot update completed travels.
        ii) If the price has changed, there will be no deduction from the users' balance who have already purchased.
        iii) If the type of vehicle is updated and if the new vehicle type does not have enough seats for travelers who purchased a seat already, then deleting the related booking and generating new booking for these users are the company's responsibility.

6) Delete purchase and refund:
    a) A company can delete a purchase and refund only if a purchase is made for one of their travels.
        i) The system checks whether purchased travel is upcoming travel and belongs to the company.
        ii) The amount of money given for the ticket is deposited back to the travelers balance.

7) Delete purchase and generate new booking:

a) A company can delete a purchase and generate a new booking only if a deleted purchase is made for one of their travels and the company can generate a new booking for one of their travels.
    i) The system checks whether deleted purchased travel is upcoming travel and belongs to the company. There will be no refund since a new booking is generated.
b) The company directed the "add purchase" page. Then, the company enters travel id, seat number, seat type and TCK of the traveler. (A scenario for adding a purchase is written already.)

8) Adding purchase:
    a) The company enters travel id, seat number, seat type and TCK of the traveler.
        i) The system checks if there is a travel whose id equals the entered id value.
        ii) The system checks whether a travel is an upcoming travel or not.
        iii) The system checks whether there is a user with the entered TCK.
        iv) The system adds a row to the Purchased relations.

3.1.3. Admin

1) Generate report:
    a) Clicking on the generate report button, admins can get the general information about the application such as total sales, total reviews, total companies etc.

2) List companies:
    a) Admins can go to the page which lists all the companies.
    b) Clicking on the "Validated Companies" will list all the validated companies ordered by validation date.
    c) Clicking on the "Unvalidated Companies" will list all the unvalidated companies.
    d) Clicking on the "All Companies" will list all companies.
    e) Admins can view detailed information about a selected company.

3) Deactivating or deleting a company:
    a) By clicking on the Deactivate button, an admin can change the active status of a company.
        i) The system checks whether this company is in the database
        ii) For the travelers who purchased an upcoming travel which is registered by the deactivated company, their money will be refunded by the system.
        iii) The system deletes all the upcoming bookings (both reservations and purchases) whose travels are registered by the deactivated company.
        iv) The system deletes all the upcoming travels which are registered by the deleted company
        v) The system should change the active status of the company.
    b) By clicking on "Delete" button, an admin can delete a company from the database.
        i) The system checks whether this company is in the database
        ii) For the travelers who purchased an upcoming trip which is registered by the deleted company, their money will be refunded by the system.
        iii) The system deletes all the bookings that are related with this company.
        iv) The system deletes all the upcoming and past travels that are registered by the company.
        v) The system deletes all the reviews for the travels which were registered by the deleted company.
        vi) The system deletes the company.

4) Validate a company:
    a) Lists all companies or all unvalidated companies.
    b) Admin clicks validate button for the company
        i) The system updated the Company relation by updating the validation date from null value to the validated day and time.

5) List travelers / see detailed information about a traveler:
    a) Admins can view all travelers and their information such as TCK, full name, age and balance.

b) Admin can select a traveler and view personal information, journeys, past and upcoming travels and comments of the selected traveler.

6) Deactivating or deleting a traveler:
    a) By clicking on the deactivate button, an admin can deactivate a traveler
        i) The system checks whether this traveler is in database
        ii) The system first deletes all reservations which are made by the deactivated traveler for upcoming travels. Reservations for past travels, purchases for past and upcoming travels will not be deleted.
        iii) The system deactivates the traveler.
    b) By clicking on the "delete" button, an admin can delete a traveler from the database.
        i) The systems checks whether this traveler is in database
        ii) The system deletes all upcoming and past bookings which includes reservations or purchases.
        iii) The system deletes all Journeys belonging to the traveler.
        iv) The system deletes coupons for the traveler.
        v) The system deletes all reviews of this user.
        vi) The system deletes the user.

7) Generating a coupon:
    a) Admin enter a coupon name, sale rate, expiration date, and public status
        i) The system checks if there exists a coupon with the same coupon name.
        ii) If there is no existing coupon with the same name, add the coupon.
        iii) If there is an existing coupon, then the system checks the expiration date of the existing coupon.
        iv) If an existing coupon has expired, the coupon is added. However, if the existing coupon hasn't expired, the admin cannot add the coupon.
8) Deleting a coupon:
    a) Admin clicks on delete button
        i) The system checks whether a coupon exists or not.
        ii) The system deletes all related tuples with in the Coupon_Traveler relation
        iii) The system sets null value for the corresponding attribute in the purchases.
        iv) The system deletes the coupon.

9) List vehicle types:
    a) List all vehicle types and view their details.

10) Adding vehicle type:
    a) An admin enters model, type, seat formation, number of seat and business class rows of the vehicle

11) Deleting a vehicle type:
    a) An admin clicks delete button:
        i) The system checks whether there is such vehicle type
        ii) The attribute corresponding to the vehicle type in travels are set to null values
        iii) The vehicle type is deleted.

12) List all terminals:

a) Admin can view all the terminals and their information.

13) Add a terminal:
    a) An admin enters terminal name, city, type.
        i) There cannot be any terminal with the same name.

14) Delete a terminal:
    a) Click on delete button for a terminal
        i) The system sets null for the attributes in the corresponding rows in the travel relation.

15) Delete purchase with or without refund:
    a) An admin can delete any purchase and refund by clicking on the delete with refund button.
        i) The system checks whether there is such purchase
        ii) The amount of money given for the ticket is deposited back to the travelers balance.
        iii) The system deletes the related purchase
    b) An admin can delete any purchase without refund by clicking on the delete without refund button.
        i) The system checks whether there is such purchase
        ii) The system deletes the related purchase

16) Delete reservation / add reservation to any travel for any traveler.
    a) Enter travel_id, seat_number, seat_type and TCK of the traveler for adding a reservation
    b) Click on the delete button for deleting a reservation.

17) Add purchase with money deduction or without money deduction for any travel and any traveler.
    a) An admin clicks on "add purchase with deduction"
        i) Enter travel_id, seat_number, seat_type and traveler TCK of the traveler.
            (1) The system adds purchase
        ii) Update traveler balance
            (1) The system makes deductions from the traveler's balance.
    b) An admin clicks on "add purchase without deduction"
        i) Enter travel_id, seat_number, seat_type and traveler TCK.
            (1) The system adds purchase

18) View / delete an review
    a) List all the reviews
    b) Apply filters on reviews.
    c) Delete an review.

3.2. Algorithms

During the implementation, there are no specific algorithms which are planned to be used. The functionalities related to database elements will be implemented using SQL queries.

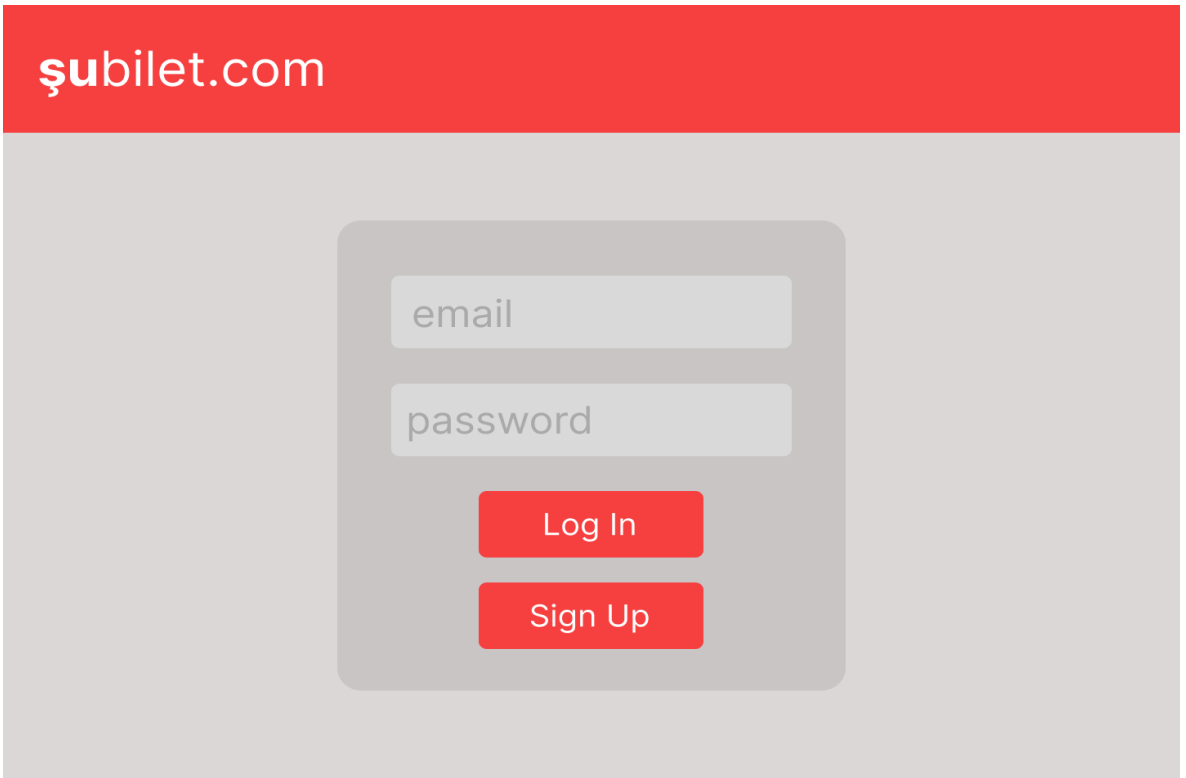
3.3. Data Structures

The data stored in the database mainly consists of three types of data: date, number, and alphabetic. Numbers will be stored with INT type. Alphanumeric values will be stored with VARCHAR type. Date values will be stored with DATETIME type.

4. UI Design and Corresponding SQL Queries

NOTE: In all SQL queries present in this section, @ sign denotes that the specified value is taken as input from the backend/frontend.

4.1. Login/Register

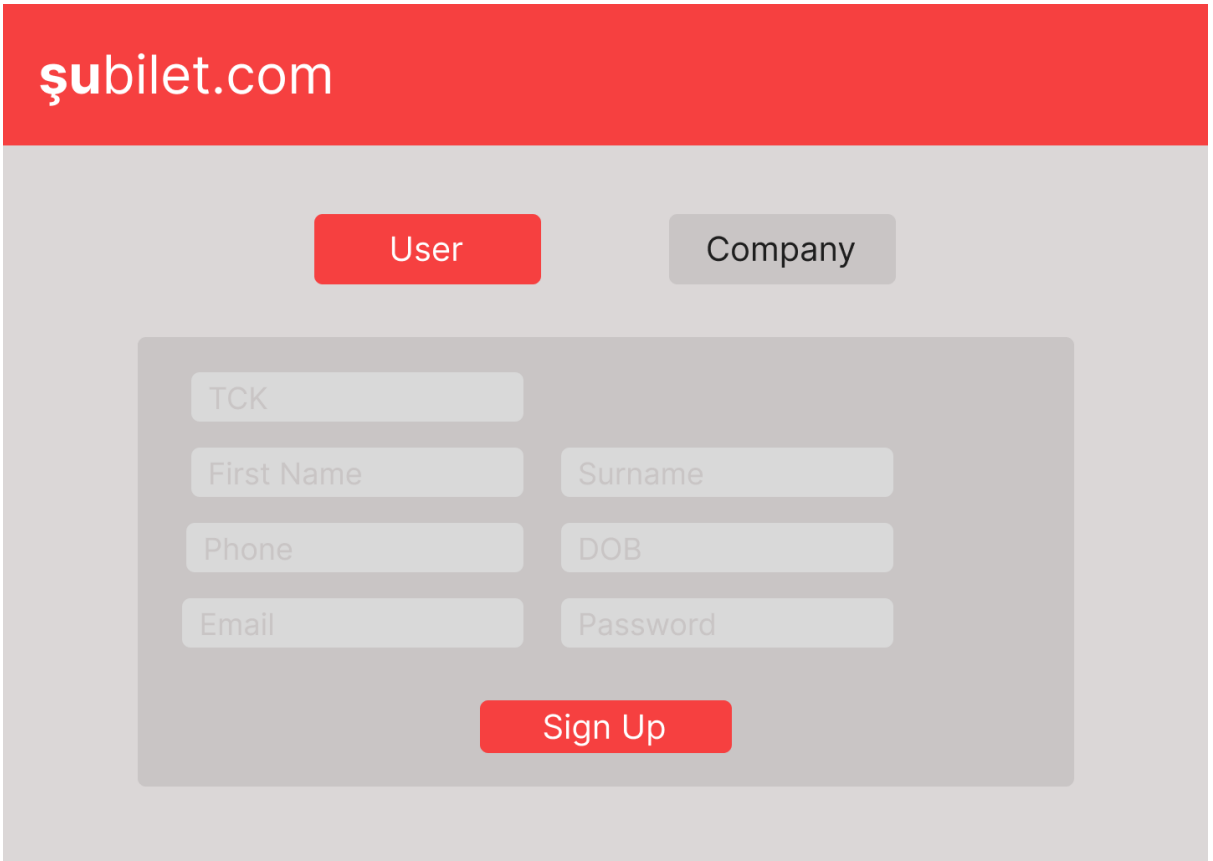


Login:

```
SELECT *
FROM User
WHERE email = @email AND password = @password
```

- If an admin logs in, then the last login time is updated for the admin

```
UPDATE Administrator
SET last-login-time = NOW()
WHERE id = ( SELECT id
             FROM User
             WHERE email = @email )
```



Register:

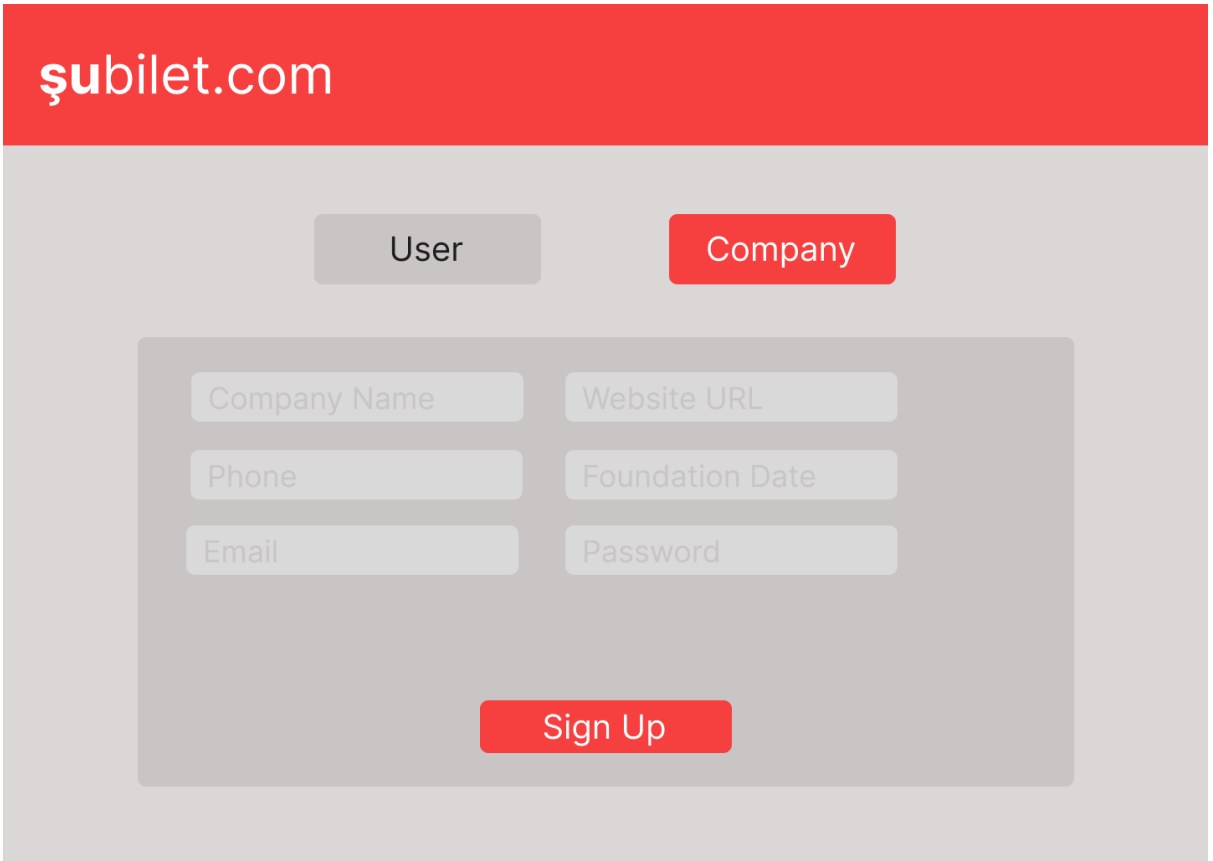
To check if a user with the specified email already exists:

```
SELECT *
FROM User
WHERE email = @email
```

- If this query returns a result, the user will be alerted to use another email.

Register as traveler:

```
INSERT INTO User(id, email, password, phone)
values(NULL, @email, @password, @phone); --NULL because ID is auto generated
INSERT INTO Traveler (id, TCK, name, surname, age, balance)
values( (SELECT id FROM User WHERE email = @email) , @TCK, @name,
@surname, @age, 0);
```



Register as company:

```
INSERT INTO User(id, email, password, phone)
values(NULL, @email, @password, @phone);
INSERT INTO Company(id, company_name, website, foundation_date, about,
validator_id, validation_time)
values( (SELECT id FROM User WHERE email = @email), @company_name,
@website, @foundation_date, @about, null, null);
```

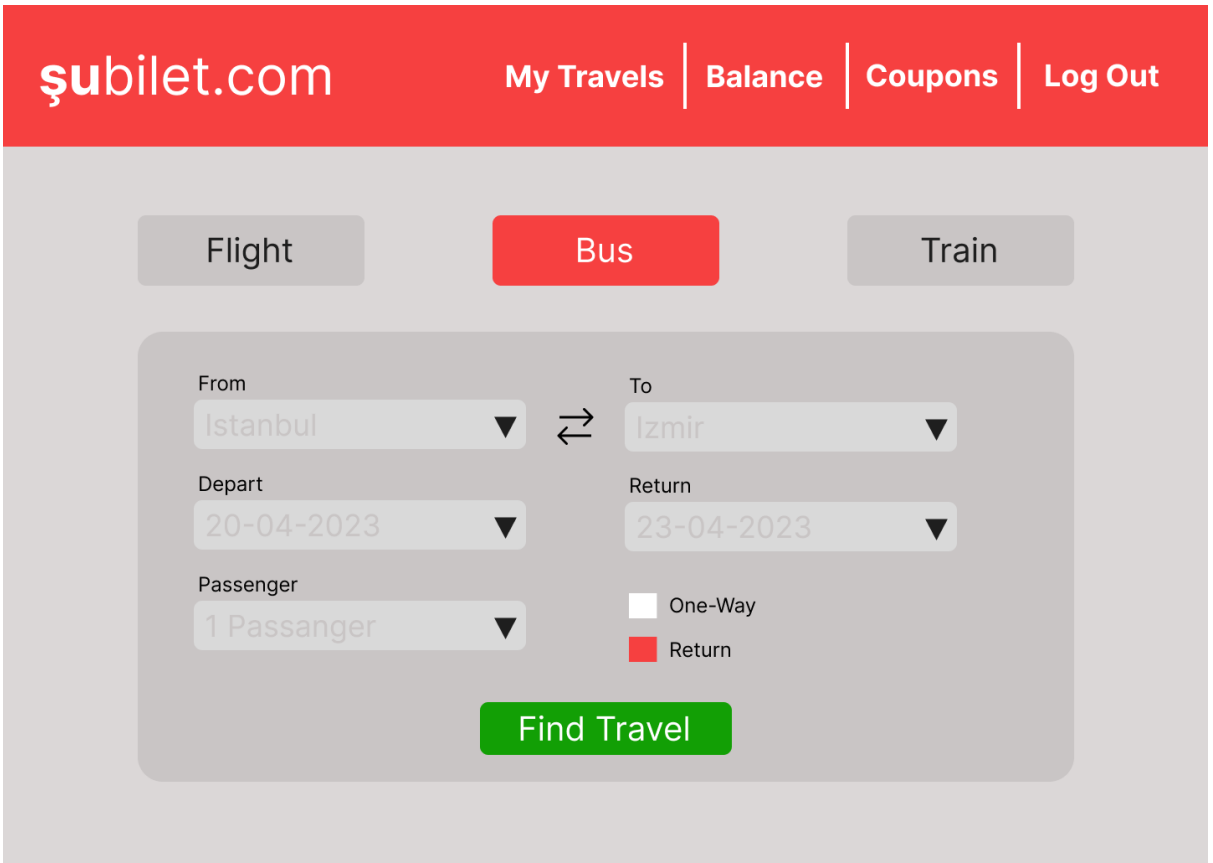
- Company will first be registered without validation, therefore validator id and validation time is null.
- When an admin validates a company, those two fields will be updated accordingly.

Register as admin:

- Admins will be manually inserted to the database with the following query:

```
INSERT INTO User(id, email, password, phone)
values(NULL, @email, @password, @phone);
INSERT INTO Administrator(id, username, last_login_time, income)
values( (SELECT id FROM User WHERE email = @email) , @username, null,
@income);
```

4.2. Buying a Ticket



Select all cities in the database (no duplicates):

```
SELECT DISTINCT city
FROM Terminal
ORDER BY city
```

Select the terminal names with their cities:

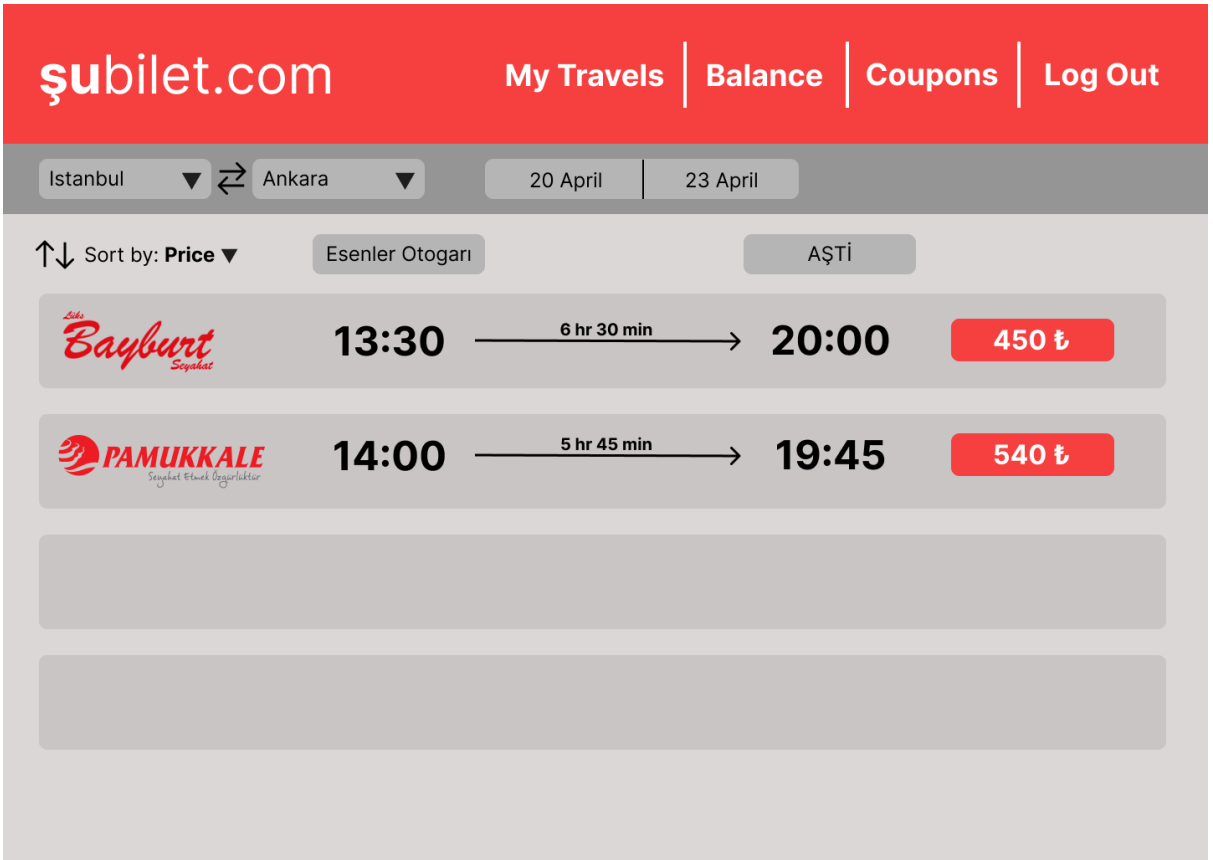
```
SELECT name, city
FROM Terminal
ORDER BY city
```

List all available travels with specified filters:

```
SELECT *
FROM Travel T join Terminal Dep on T.departure_terminal_id =
Dep.terminal_id join Terminal Ar on T.arrival_terminal_id = Ar.terminal_id
join Vehicle_Type V on V.id = T.vehicle_type_id
WHERE Dep.city = @departs_from, Ar.city = @arrives_to, T.depart_time =
@departure_time, T.arrive_time = @arrive_time, V.type = @vehicle_type
```

- This returns all possible travels from the specified departure city to arrival city, on the specified departure day, with the specified travel type(plane, bus, etc.)

- Vehicle type is retrieved from buttons on the top of the page



Check if the chosen travel's departure/arrival time collides with any of the user's travels' departure/arrival time:

Check departure:

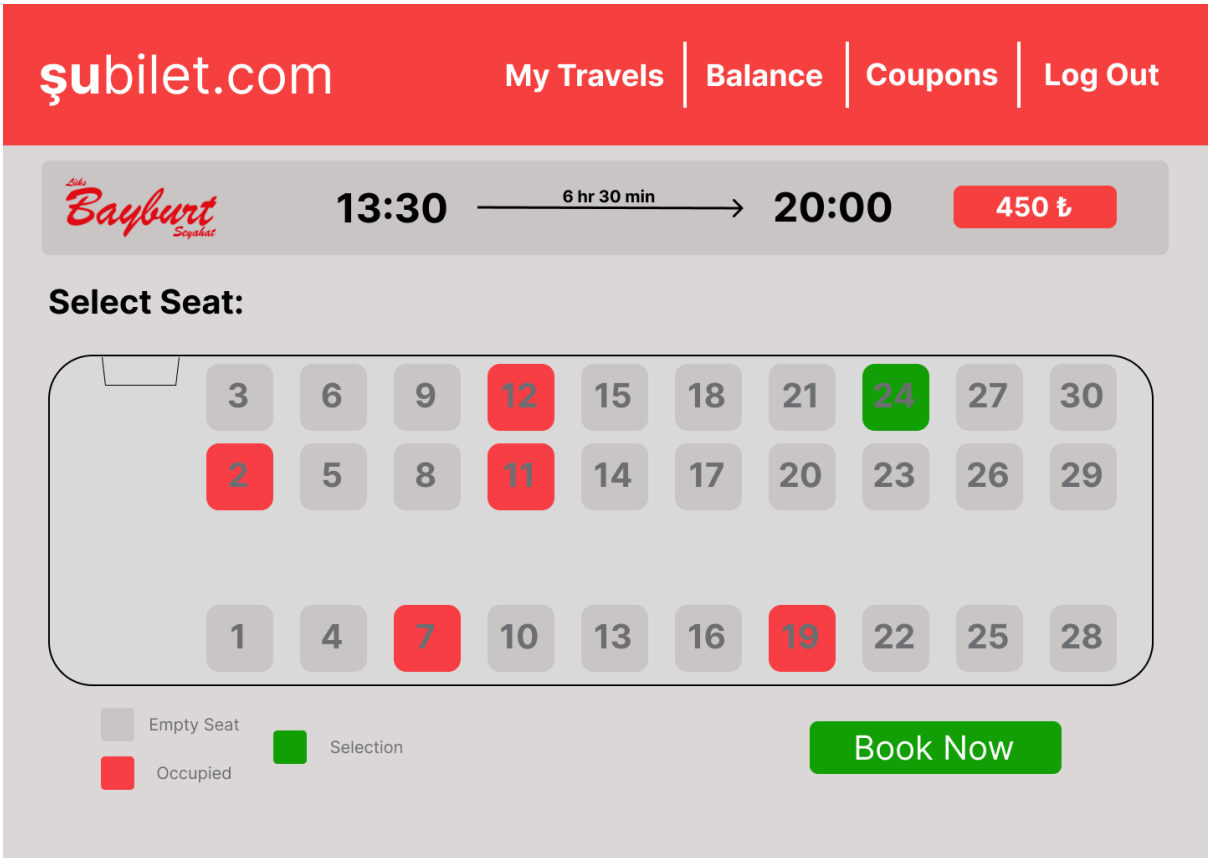
```
SELECT *
FROM Booking B natural join Travel T join Traveler TP on TP.id =
Booking.traveler_id
WHERE TP.id = @user_id AND
T.depart_time BETWEEN (SELECT depart_time FROM Travel WHERE travel_id =
@ID_of_chosen_travel) AND (SELECT arrive_time FROM Travel WHERE travel_id
= @ID_of_chosen_travel) -- this is a scalar subquery, so "=" can be used
```

Check arrival:

```
SELECT *
FROM Booking B natural join Travel T join Traveler TP on TP.id =
Booking.traveler_id
WHERE TP.id = @user_id AND
```

```
T.arrival_time BETWEEN (SELECT depart_time FROM Travel WHERE travel_id =
@ID_of_chosen_travel) AND (SELECT arrive_time FROM Travel WHERE travel_id
= @ID_of_chosen_travel)
```

- In both cases, the user will be notified; however, they will still be able to continue and purchase the ticket if they wish so.



Get seat formation information, so the user can choose their seat:

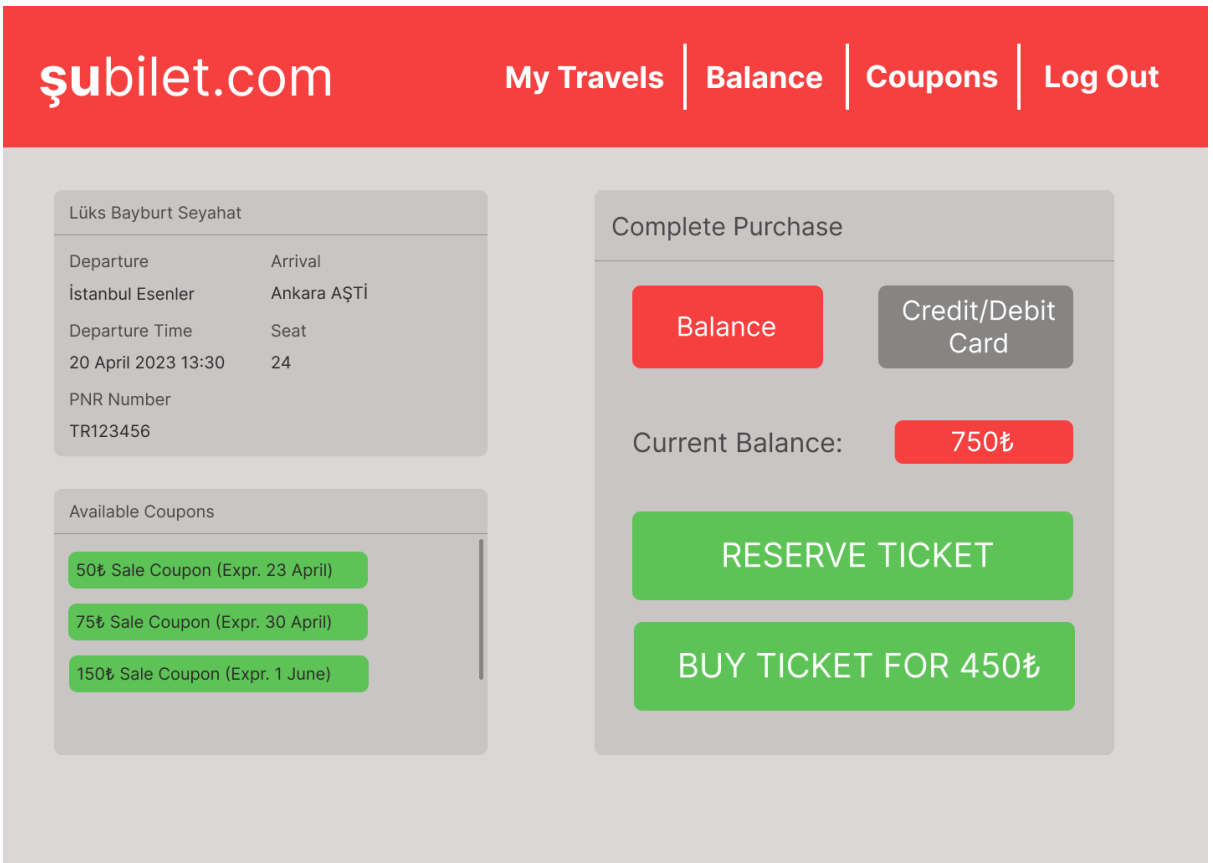
Get all seats of the travel type:

```
SELECT seat_formation, num_of_seats, business_rows
FROM VehicleType
WHERE id = (SELECT vehicle_type_id FROM Travel WHERE travel_id =
@chosen_travel);
```

Get the seat numbers of unavailable seats (already purchased or reserved):

```
SELECT seat_number
FROM Booking
WHERE travel_id = @travel_id
```

- This seat information is retrieved as raw data (number of seats, columns, gaps etc.) It will be utilized in the frontend to display the available seats to the user.



The traveler purchases the ticket with their chosen payment method:

If the user selected a coupon, check if the coupon is expired:

```
SELECT *
FROM Sale_Coupon
WHERE coupon_id = @coupon_id AND expiration_date < NOW()
```

- If this query returns any results, it would mean that the coupon which the user is applying has expired, which would then display an error message.

```
INSERT INTO Booking(PNR, travel_id, seat_number, traveler_id)
values(@PNR, @travel_id, @seat_number, @traveler_id);

INSERT INTO Purchased(PNR, purchased_time, payment_method, price,
coupon_id)
values(@PNR, NOW(), @payment_method, @coupon_id, (SELECT price FROM Travel
WHERE travel_id = @travel_id)
);
```

- If the traveler uses a coupon during purchase:

```
UPDATE Traveler
```

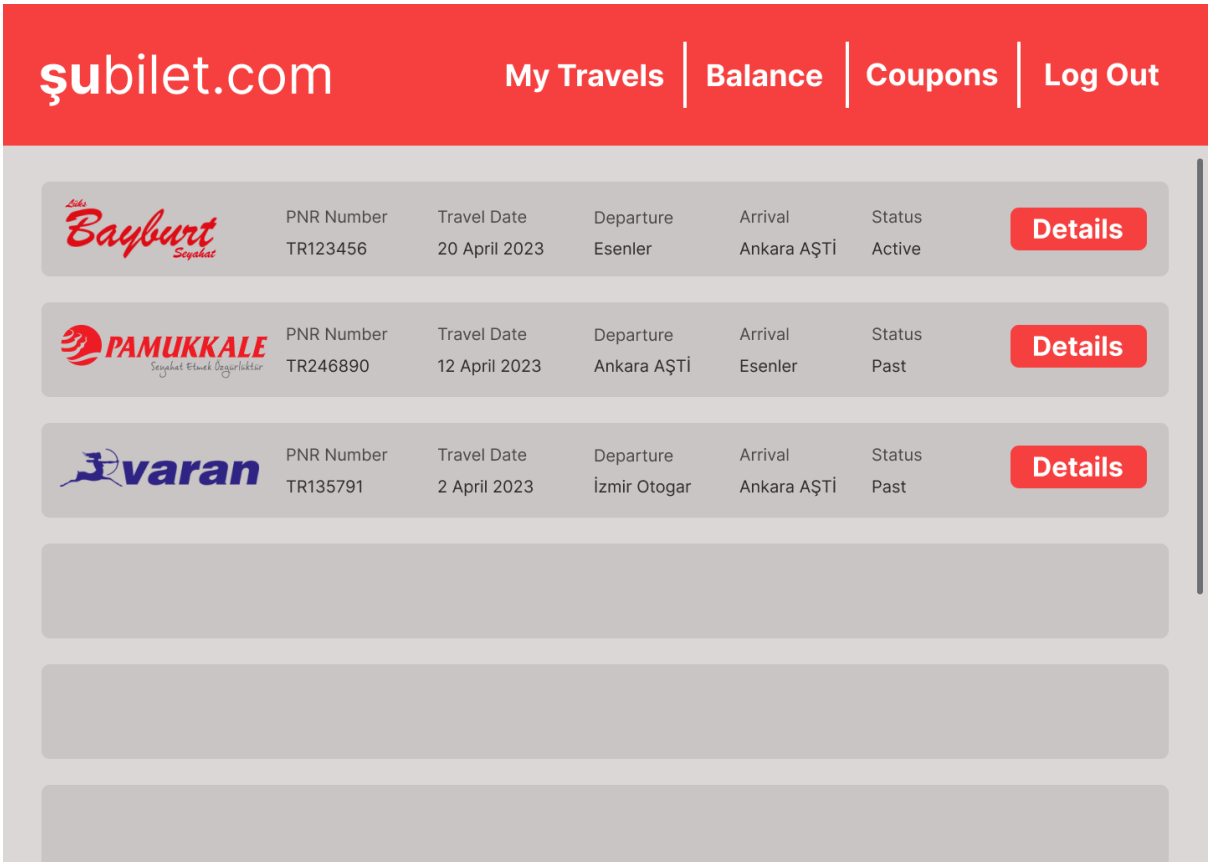
```
SET balance = balance - (SELECT price FROM Purchased WHERE PNR =
@PNR)*(SELECT sale_rate FROM Sale_Coupon WHERE coupon_id = @coupon_id)
WHERE Traveler.id = @traveler_id;
```

- If the traveler doesn't use a coupon during purchase:

```
UPDATE Traveler
SET balance = balance - (SELECT price FROM Purchased WHERE PNR = @PNR)
WHERE Traveler.id = @traveler_id;
```

- The available coupons for the user are listed.

```
SELECT *
FROM Coupon_Traveler natural join Sale_Coupon
WHERE user_id = @user_id AND used_status = False AND expiration_date >
NOW()
ORDER BY expiration_date ASC;
```



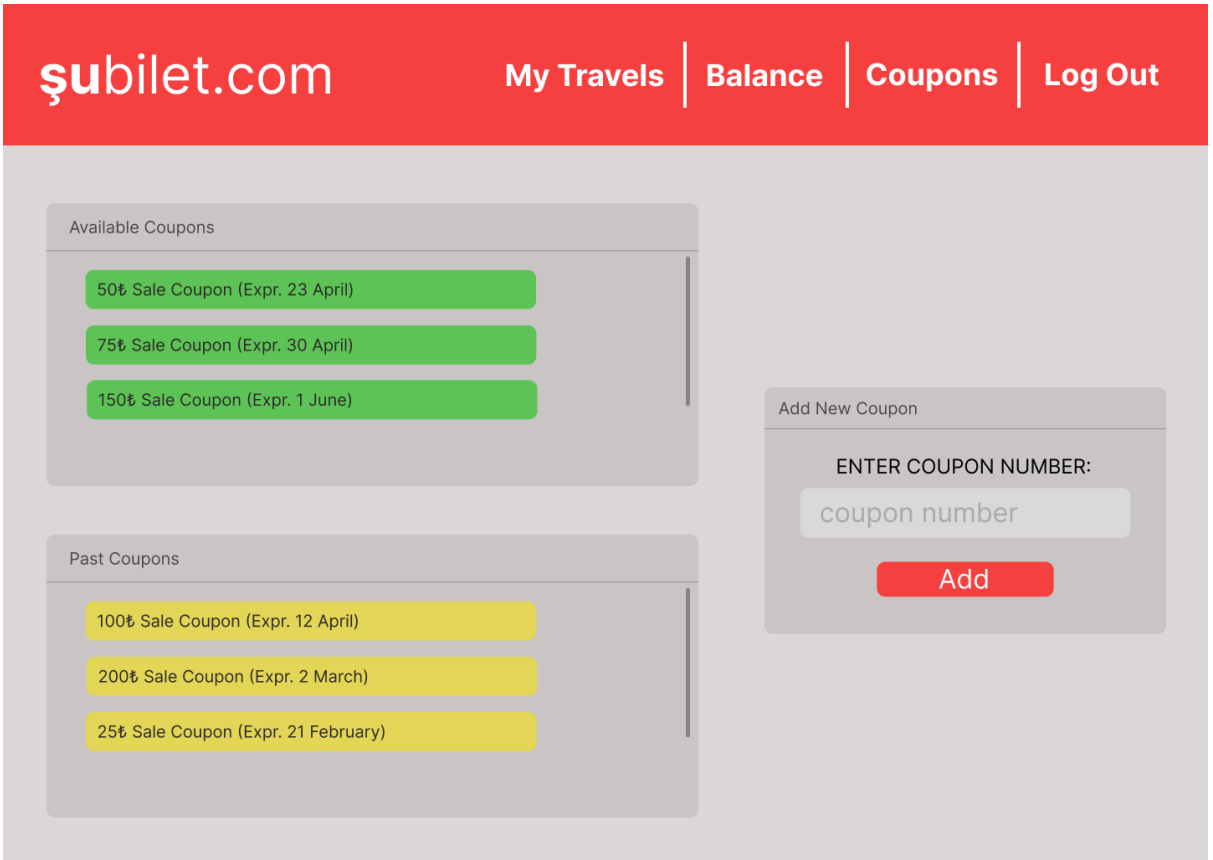
Show the travel list of the traveler (upcoming and past travels), ordered by descending departure time:

```
SELECT *
FROM Booking natural join Travel
```

```
WHERE traveler_id = @traveler_id
ORDER BY depart_time desc;
```

- This query selects all attributes from the database, but only the relevant information will be displayed in the UI, as seen in the image.

4.3. Coupon



Get user's available (not used yet) coupons:

```
SELECT *
FROM Coupon_Traveler natural join Sale_Coupon
WHERE user_id = @user_id AND used_status = False AND expiration_date >
NOW()
ORDER BY expiration_date ASC;
```

Get user's past (used) coupons with the information of which travels they were used in:

```
WITH Users_Purchased_Travel_withLocation as (
SELECT coupon_id, Purchased.price as price, depart_time, T1.city as
dep_city, T2.city as arr_city
FROM Purchased natural join Booking join Travel on Booking.travel_id =
Travel.travel_id join Terminal T1 on departure_terminal_id =
Travel.terminal_id join Terminal T2 on arrival_terminal_id =
Travel.terminal_id
WHERE traveler_id = @user_id)
SELECT coupon_name, price, depart_time, dep_city, arr.city
FROM Users_Purchased_Travel_withLocation natural join Sale_Coupon
WHERE coupon_id IS NOT null
```

User adds a coupon:

```
INSERT INTO Coupon_Traveler(coupon_id, user_id, used_status)
values ( (SELECT coupon_id FROM Sale_Coupon SC WHERE SC.coupon_name =
@entered_coupon_name AND SC.expiration_date > NOW()), @user_id, FALSE)
```

First the system needs to check whether there is a coupon with the entered name, and if there is a coupon with this name, it should not have expired.

```
SELECT *
FROM Sale_Coupon SC
WHERE SC.coupon_name = @entered_coupon_name AND SC.expiration_date > NOW()
```

- If this query returns any results, it would mean that there is a coupon with the entered name which has not expired yet. So, it can be added if the traveler hasn't added this coupon yet. Otherwise error message indicating there is no available coupon.

For public or private coupons, check if the user has already added the said coupon:

```
SELECT *
FROM Sale_Coupon
WHERE coupon_id = @entered_coupon_id AND coupon_id IN (SELECT coupon_id
FROM Coupon_Traveler where user_id = @user_id)
```

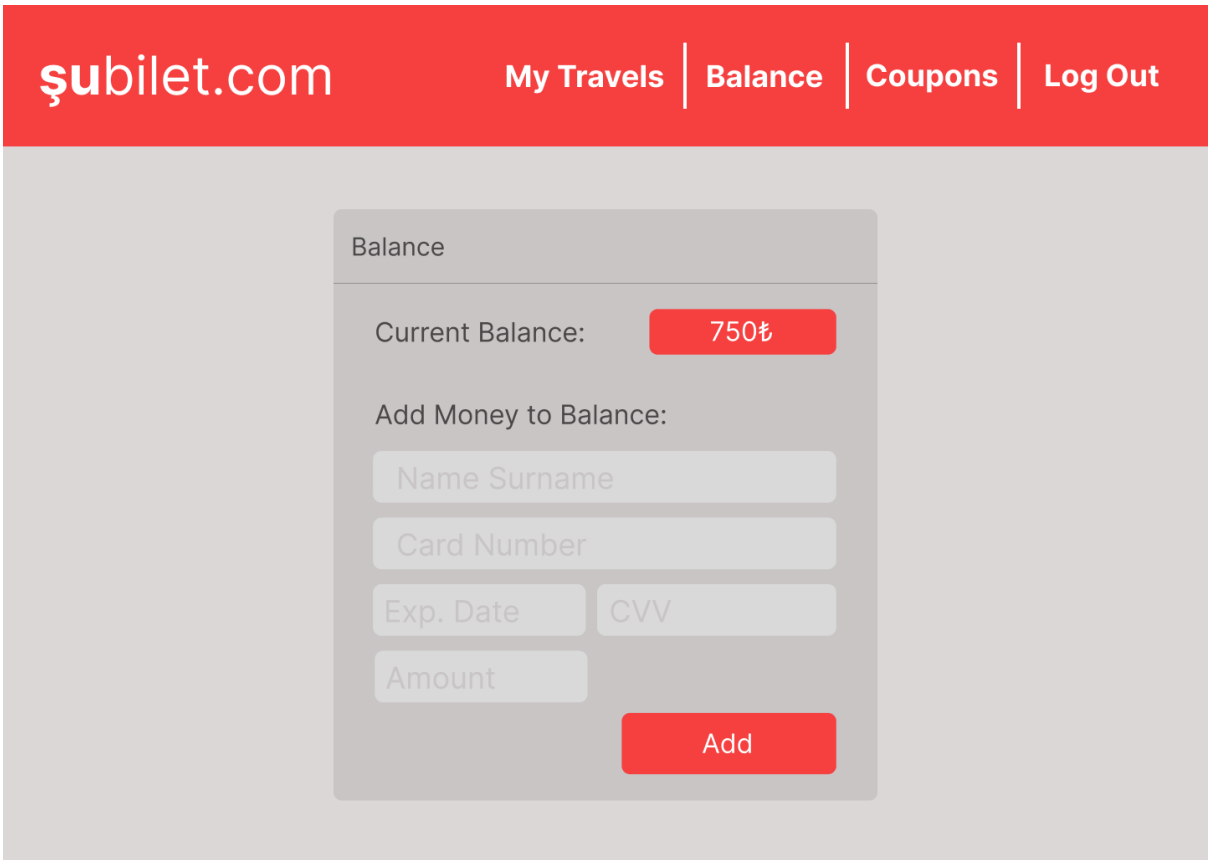
- If this query returns any results, it would mean that the user is trying to add a coupon that they have already added before. Therefore, a corresponding error message would be displayed to the user.

For private coupons, check if another user has added the said coupon:

```
SELECT *
FROM Sale_Coupon
WHERE coupon_id = @entered_coupon_id AND public_status = "private" AND
coupon_id IN (SELECT coupon_id FROM Coupon_Traveler)
```

- If this query returns any results, it would mean that the user is trying to add a private coupon that was already added by another user, which is not allowed in private coupons. Corresponding error message would be displayed.

4.4. Balance



Get user's balance:

```
SELECT balance
FROM Traveler
WHERE user_id = @user_id
```

Add money to user's balance:

```
UPDATE Traveler
SET balance = balance + @newAmount
WHERE user_id = @user_id
```

5. Advanced Database Components

5.1. Views

A view of upcoming travels:

```
CREATE VIEW upcoming-travels AS
SELECT *
FROM Travel
WHERE NOW() < depart-time
```

A view of travelers from the perspective of a company:

```
CREATE VIEW company-traveler-view AS
SELECT TCK, name, surname, age, email, phone
FROM  User JOIN Traveler
      ON User.id = Traveler.id;
```

5.2. Assertions

Assert that the chosen seat number is not larger than the total number of seats in a vehicle:

```
CREATE ASSERTION purchased-seat-number
CHECK ( NOT EXISTS
      ( SELECT *
        FROM    Booking B JOIN Travel T
            ON  B.travel_id = T.travel_id
       JOIN Vehicle_Type V
            ON  T.vehicle_type_id = V.id
        WHERE  V.num_of_seats < B.seat_number ));
```

Assert that a coupon used in a purchase is correctly set as "Used":

```
CREATE ASSERTION coupon-use-status
CHECK ( NOT EXISTS
      ( SELECT *
        FROM   Booking B JOIN Purchased P
            ON  B.PNR = P.PNR
        JOIN Coupon_Traveler C
            ON  C.user_id = B.traveler_id
              AND  P.coupon_id = C.coupon_id
        WHERE C.used_status = FALSE ));
```

Assert that different types of users can not have the same ID:

```
CREATE ASSERTION user-specification
CHECK ( NOT EXISTS (SELECT *
                     FROM Traveler T, Company C
                     WHERE T.id = C.id)
          AND NOT EXISTS (SELECT *
                      FROM Traveler T, Admin A
                      WHERE T.id = A.id)
          AND NOT EXISTS (SELECT *
                      FROM Admin A, Company C
                      WHERE A.id = C.id));
```

5.3. Triggers

- **Purchase-reservation**

```
CREATE TRIGGER purchase-on-reserve
      BEFORE INSERT ON Purchased
REFERENCING NEW ROW AS new-row
FOR EACH ROW
      DELETE FROM Booking B
      WHERE EXIST ( SELECT *
                       FROM Reserved R
                       WHERE R.PNR = B.PNR AND
                       new-row.PNR = R.PNR);
```

- **When a company is deactivated from the admin, price of purchased tickets of all travels which are registered by the deactivated company will be refund to the users**

```
CREATE TRIGGER deactivate-company
      AFTER UPDATE OF User ON (active)
REFERENCING OLD ROW AS old-row
REFERENCING NEW ROW AS new-row
FOR EACH ROW
WHEN new-row.active = FALSE AND old-row.active = TRUE
      AND old-row.id IN ( SELECT id
                          FROM Company)
BEGIN
      WITH  inactive_travels (travel_id)  AS
                 ( SELECT travel_id
                   FROM   upcoming-travels U
                   WHERE  U.travel_company_id = old-row.id )
```

```
      DECLARE current_travel_id INT;
      DECLARE travel_cursor CURSOR FOR SELECT travel_id FROM
inactive_travels;

      -- Open cursor and iterate over inactive_travels
      OPEN travel_cursor;

      travel_loop: LOOP
            FETCH travel_cursor INTO current_travel_id;
            CALL refund_travel(current_travel_id);
      END LOOP;

      CLOSE travel_cursor;

      DELETE FROM Travel
      WHERE Travel.travel_id  IN  (SELECT travel_id
                                   FROM inactive_travels)

END;
```

- **The trigger below gives the same result with the trigger above without using cursors and procedures.**

```
CREATE TRIGGER deactivate-company
      AFTER UPDATE OF User ON (active)
REFERENCING OLD ROW AS old-row
REFERENCING NEW ROW AS new-row
FOR EACH ROW
WHEN new-row.active = FALSE AND old-row.active = TRUE
      AND old-row.id IN ( SELECT id
                          FROM Company)
BEGIN
      WITH  inactive_travels (travel_id)  AS
                  ( SELECT travel_id
                    FROM   upcoming-travels U
                    WHERE  U.travel_company_id = old-row.id ),
            refund_amount (traveler_id, total_refund) AS
                  ( SELECT B.traveler_id, SUM(P.price)
                    FROM Booking B JOIN inactive_travels IT
                        ON B.travel_id = IT.travel_id
                    JOIN Purchased P
                        ON P.PNR = B.PNR
                    GROUP BY B.traveler_id )
      BEGIN
            UPDATE traveler T2
            SET T2.balance = T2.balance + (SELECT total_refund
```

```
                                        FROM refund_amount R
                                        WHERE R.traveler_id = T2.id)
            WHERE T2.id  IN  (SELECT traveler_id
                              FROM refund_amount) ;

            DELETE FROM Travel
            WHERE Travel.travel_id  IN  (SELECT travel_id
                                         FROM inactive_travels)
      END
END;
```

- **When an admin deletes a company, the trigger below, which is named as before-delete-company, changes the activation status of this company first. Changing the activation status of the company will trigger another trigger named deactivate-company.**

```
CREATE TRIGGER before-delete-company
      BEFORE DELETE ON User
REFERENCING OLD ROW AS old-row
WHEN old-row.active = TRUE AND old-row.id IN ( SELECT id
                                               FROM COMPANY)
UPDATE User
SET old-row.active = FALSE;
```

- **When a user is deactivated by admin, all reservations which are made by the deactivated traveler for upcoming travels are deleted from the database.**

```
CREATE TRIGGER deactivate-traveler
      BEFORE UPDATE OF User ON (active)
REFERENCING OLD ROW AS old-row
REFERENCING NEW ROW AS new-row
FOR EACH ROW
WHEN new-row.active = FALSE AND old-row.active = TRUE
      AND old-row.id IN ( SELECT id
                          FROM Traveler)
DELETE FROM Booking B
WHERE  B.PNR IN (SELECT R.PNR
                      FROM Reserved R)
                AND B.traveler_id = old-row.traveler_id
                AND  B.travel_id  IN ( SELECT travel_id
                                       FROM upcoming-travels); --upcoming-travels
is a view
```

- **When a departure or arrival time of a travel changes, the validity of the journeys that includes that travel must be checked.**

```
CREATE TRIGGER travel-schedule-change
      AFTER UPDATE OF Travel ON (depart_time, arrive_time)
REFERENCING NEW ROW AS new-row
UPDATE Journey J
SET isValid = (CASE
                  WHEN
                        (EXISTS (SELECT *
                                 FROM Travels_In_Journey TJ1, Travel T1,
                                    Travels_In_Journey TJ2, Travel T2
                                 WHERE TJ1.journey_name = TJ2.journey_name
                                    AND TJ1.traveler_id = TJ2.traveler_id
                                    AND TJ1.journey_name = J.journey_name
                                    AND TJ1.traveler_id = J.traveler_id
                                    AND TJ1.travel_id <> TJ2.travel_id
                                    AND TJ1.travel_id = T1.travel_id
                                    AND TJ2.travel_id = T2.travle_id
                                    AND ((T1.depart_time BETWEEN
                                    T2.depart_time AND T2.arrive_time) OR
                                    (T2.depart_time BETWEEN T1.depart_time
                                    AND T1.arrive_time)))
                        THEN “not valid”
                     ELSE “valid”
                  END);
WHERE EXISTS(SELECT *
             FROM Travels_In_Journey TJ
             WHERE TJ.journey_name = J.journey_name,
                   TJ.traveler_id = J.traveler_id,
                   travel_id = new-row.travel_id)
```

- **When a travel is added to a journey, the validity of this journey must be checked.**

```
CREATE TRIGGER check-journey-after-travel-addition
      AFTER INSERT ON Travels_In_Journey
REFERENCING NEW ROW AS new-row
FOR EACH ROW
WHEN new-row.travel_id NOT IN
      ( SELECT travel_id
      FROM Travels_In_Journey TJ1
      WHERE TJ1.journay_name = new-row.journay_name
            AND TJ1.traveler_id = new-row.traveler_id)
BEGIN
UPDATE Journey J
```

```
SET isValid = (CASE
                  WHEN
                        (EXISTS (SELECT *
                                 FROM Travels_In_Journey TJ1, Travel T1,
                                    Travels_In_Journey TJ2, Travel T2
                                 WHERE TJ1.journey_name = TJ2.journey_name
                                    AND TJ1.traveler_id = TJ2.traveler_id
                                    AND TJ1.journey_name = J.journey_name
                                    AND TJ1.traveler_id = J.traveler_id
                                    AND TJ1.travel_id <> TJ2.travel_id
                                    AND TJ1.travel_id = T1.travel_id
                                    AND TJ2.travel_id = T2.travle_id
                                    AND ((T1.depart_time BETWEEN
                                    T2.depart_time AND T2.arrive_time) OR
                                    (T2.depart_time BETWEEN T1.depart_time
                                    AND T1.arrive_time)))
                        THEN “not valid”
                   ELSE “valid”
                END);
WHERE new-row.journay_name = J.journey_name
      AND new-row.traveler_id = J.traveler_id
END
```

- **The trigger below, which gives the same result as the one above, uses prescribed procedure.**

```
CREATE TRIGGER check-journey-after-travel-addition
      AFTER INSERT ON Travels_In_Journey
REFERENCING NEW ROW AS new-row
FOR EACH ROW
WHEN new-row.travel_id NOT IN
      ( SELECT travel_id
      FROM Travels_In_Journey TJ1
      WHERE TJ1.journay_name = new-row.journay_name
            AND TJ1.traveler_id = new-row.traveler_id)
BEGIN
      DECLARE journay_valid BIT;
      CALL isJourneyValid(new-row.journey_name, new-row.traveler_id,
journey_valid);
      UPDATE Journey J
      SET J.isValid = CASE
                        WHEN journey_valid = 1 then 'valid'
                        ELSE “not valid”
                        END
```

```
      WHERE new-row.journay_name = J.journey_name
            AND new-row.traveler_id = J.traveler_id
END
```

- **When a travel is deleted to a journey, the validity of this journey must be checked. The below SQL trigger is written without using any procedure.**

```
CREATE TRIGGER check-journey-after-travel-addition
      AFTER DELETE ON Travels_In_Journey
REFERENCING OLD ROW AS old-row
FOR EACH ROW
BEGIN
UPDATE Journey J
SET isValid = (CASE
                  WHEN
                        (EXISTS (SELECT *
                                 FROM Travels_In_Journey TJ1, Travel T1,
                                    Travels_In_Journey TJ2, Travel T2
                                 WHERE TJ1.journey_name = TJ2.journey_name
                                    AND TJ1.traveler_id = TJ2.traveler_id
                                    AND TJ1.journey_name = J.journey_name
                                    AND TJ1.traveler_id = J.traveler_id
                                    AND TJ1.travel_id <> TJ2.travel_id
                                    AND TJ1.travel_id = T1.travel_id
                                    AND TJ2.travel_id = T2.travle_id
                                    AND ((T1.depart_time BETWEEN
                                    T2.depart_time AND T2.arrive_time) OR
                                    (T2.depart_time BETWEEN T1.depart_time
                                    AND T1.arrive_time)))
                        THEN “not valid”
                    ELSE “valid”
                END);
WHERE old-row.journay_name = J.journey_name
      AND old-row.traveler_id = J.traveler_id
END
```

- **When a travel is deleted to a journey, the validity of this journey must be checked. The SQL trigger written below results in the one written above whose name is also "check-journey-after-travel-addition". This trigger uses a procedure which is called "isJourneyValid".**

```
CREATE TRIGGER check-journey-after-travel-addition
      AFTER DELETE ON Travels_In_Journey
REFERENCING OLD ROW AS old-row
FOR EACH ROW
BEGIN
      DECLARE journey_valid BIT;
      CALL isJourneyValid(old-row.journey_name, old-row.traveler_id,
journey_valid);
      UPDATE Journey J
      SET J.isValid = CASE
                        WHEN journey_valid = 1 THEN 'valid'
                        ELSE 'not valid'
                      END
      WHERE J.journey_name = old-row.journey_name
            AND J.traveler_id = old-row.traveler_id;
END;
```

5.4. Stored Procedures

- **The refund_travel procedure is used in triggers for refunding money to travelers in case of deletion of a company or travel etc.**

```
CREATE PROCEDURE refund_travel ( IN refund_travel_id INT)
BEGIN
      UPDATE traveler T
      SET T.balance = T.balance + (SELECT P.price
                                   FROM Booking B1 JOIN Purchased P
                                      ON B1.PNR = P.PNR
                                   WHERE  B1.travel_id = refund_travel_id)
      WHERE T.traveler_id  IN (SELECT B2.traveler_id
                               FROM Booking B2
                               WHERE B2.travel_id = refund_travel_id
                                    AND B2.PNR IN (SELECT PNR
                                                   FROM Purchased)
END;
```

- **The isJourneyValid procedure checks time collisions for trips in the journey.**

```
CREATE PROCEDURE isJourneyValid ( IN in_journey_name CHAR(256), IN
in_travler_id INT, OUT isValid BIT)
BEGIN
      SELECT IF( NOT EXISTS (SELECT *
                     FROM Travels_In_Journey TJ1, Travel T1,
                     Travels_In_Journey TJ2, Travel T2
                     WHERE TJ1.journey_name = TJ2.journey_name
                        AND TJ1.traveler_id = TJ2.traveler_id
                        AND TJ1.journey_name = in_journey_name
                        AND TJ1.traveler_id = in_traveler_id
                        AND TJ1.travel_id <> TJ2.travel_id
                        AND TJ1.travel_id = T1.travel_id
                        AND TJ2.travel_id = T2.travle_id
                        AND ((T1.depart_time BETWEEN T2.depart_time AND
                        T2.arrive_time) OR (T2.depart_time BETWEEN
                        T1.depart_time AND T1.arrive_time)), 1, 0)
      INTO isValid;
END;
```

6. Implementation Plan

- For the database system of the implementation, MySQL is planned to be used.
- For the programming of the implementation, Python with Flask is planned to be used, similar to the Homework 4 of this course.
- For the user interface, HTML/CSS with Flask's Jinja template is planned to be used, similar to the Homework 4 of this course.
- For virtualization of several components, Docker is planned to be used.