BEMM459

Database Technologies for Business Analytics

Team Members:-

Shashank R - 730074631

Shivani Sankannavar - 730086009

Sumukh Nagesh Bharadwaj- 730064965

Vaibhavi Ramesh - 730074762

GIT Link:- https://github.com/UniversityExeterBusinessSchool/bemm459-cw-2024-team_blr.git

YouTube Link:- https://www.youtube.com/watch?v=eYZ-9rMPpUI

Business Plan:

Air Buddy is an innovative application set to transform the way sports teams manage their travel logistics. It offers a comprehensive platform tailored specifically for the unique needs of sports teams, ensuring a seamless and stress-free travel experience. By efficiently handling all aspects of travel planning and management, Air Buddy aims to become the preferred choice for sports teams globally.

Key Features:

Centralized Platform: Serves as a centralized hub where sports teams can manage all their travel arrangements in one place. This eliminates the hassle of dealing with multiple tools and systems, streamlining the entire process.

Personalized Service: Unlike generic travel management solutions, it understands the distinct requirements of sports teams. It provides personalized recommendations and customized travel itineraries based on the team's preferences and schedules.

Efficiency and Convenience: Automates various tasks such as itinerary planning, bulk booking of flights and accommodations, and handling special requests. This automation not only saves valuable time but also ensures accuracy and consistency in travel arrangements.

Transparency and Accountability: Promotes transparency and accountability throughout the travel management process. Teams have access to comprehensive booking summaries and can provide feedback on their travel experiences, ensuring that their preferences and concerns are addressed promptly.

Table:- Describing different business functions

Business Function	Description
Player Management	Manage player profiles, including personal information
Ticket Booking	Facilitate booking of flight tickets for players
Room Booking	Manage room reservations for team members during tours
Meal Preferences	Collect and manage dietary preferences and restrictions for team members, including meal preferences and food allergies
Flights	Collection of data of all the available flights
Special Requests	Handle special requests and accommodations
Booking Summary	Provide a summary of all bookings and arrangements made for each tour, including flights, accommodations, tickets, and special requests
Feedback	Collect feedback from team members regarding their travel experiences, accommodations, and overall satisfaction

Use Case Scenario : -

Let us consider an international cricket team (England cricket team) wishes to travel to another country for their upcoming games against the hosts during summer. In this scenario there might be variety of preferences by multiple players such as,

1.Player(s) wanting to select a room by seaside if available.

2.Preferring only vegetarian meal during their travel.

3.Selecting a particular Airline as they might be a loyal customer.

4.Have certain special requests throughout the tour in terms of baggage or accommodation etc. As an application, the responsibility of Air Buddy is to take care of all the conditions and favours requested by a team in an efficient manner and use the **CRUD**-Create, Read, Update and Delete operations on the tables in their database appropriately to make sure all the necessary conditions are met.

Justification of Database Choices:

For the chosen sports travel management application, the selection of databases plays a crucial role in ensuring efficient data storage, retrieval, and scalability. As there is an adaption of Polyglot persistence, structured and unstructured database management systems are used. Here's a justification for using a Relational Database Management System (RDBMS) for structured data and one NoSQL database for unstructured data:

1. Relational Database Management System (RDBMS):

RDBMS like MySQL with the help of Python are well-suited for storing structured data, such as player profiles, flight ticket bookings, and room options, as these data types have well-defined relationships and can be organized into tables with rows and columns.

Normalization: Relational databases support normalization techniques, which help eliminate data redundancy and maintain data integrity by organizing data into smaller, related tables. This facilitates efficient storage and retrieval of structured data while minimizing storage space. To consider with an example, the Room details are separated into two smaller related tables which are the room_options and room_booking related by the Room_ID attribute. This helps in keeping the room options information available even when there are not room booking yet making sure that there is no loss of wanted information.

Data relationship: Relational databases allow the establishment of relationships between tables through primary and foreign key constraints. This enables the representation of complex relationships between structured data entities, such as one-to-one, one-to-many, or many-to-many relationships. A player can have multiple room bookings over a period but each room booking belongs to one particular player. In our database, the Player table is related to the Room_Booking table on a One-to-Many relationship.

Data Integrity: RDBMS ensures data integrity through features like ACID i.e. Atomicity, Consistency, Isolation, Durability compliance and foreign key constraints, which are crucial for maintaining the consistency and reliability of critical data, such as player information and booking details.

Scalability: While traditional RDBMS may have limitations in scaling horizontally, they can handle vertical scalability efficiently, making them suitable for applications with moderate data volumes and predictable access patterns, such as entire sports travel management.

Query Flexibility: RDBMS provide powerful SQL-based query languages, allowing for complex queries and efficient retrieval of relational data, which is essential for generating reports, analysing travel trends, and managing bookings in a very simple way.

2. NoSQL Database:

Choice: MongoDB

Unstructured Data: NoSQL databases like MongoDB excel in storing unstructured or semi-structured data, in this scenario the feedback comments and special requests, as they offer flexible designs that can accommodate diverse data formats without sacrificing performance.

Document-Oriented Storage: MongoDB's document-oriented storage model allows for storing data in JSON-like documents, which closely aligns with the hierarchical nature of data in sports travel management applications, where each document can represent a feedback entry, or special request.

Horizontal Scalability: NoSQL databases are highly scalable and can easily handle large volumes of data and high throughput, making them suitable for applications experiencing rapid growth or unpredictable access patterns, such as collecting feedback from many players multiple times over a period.

High Availability: NoSQL databases typically offer built-in replication and sharding capabilities, ensuring high availability and fault tolerance, which are essential for ensuring uninterrupted access to data and maintaining service uptime, especially during peak travel periods.

Assumptions:-

Let us assume to have used Redis instead of MongoDB for this scenario. Some of the advantages about a Redis database would be:

1. Redis stands out for its speed, crucial in sports environments where swift access to player feedback and special requests is vital.
2. Redis's flexible data model accommodates diverse data structures, making it adept at handling varied player feedback formats and special requests. Whether it's text-based feedback, detailed preference lists, or multimedia attachments like photos or videos, Redis offers versatile storage options to suit the nuances of sports team interactions.

Some of the disadvantages and the reason for us choosing MongoDB over Redis would be:-

1. Limited scaling due to RAM constraints.
2. Limited querying capabilities compared to MongoDB and this lacks support to un complex queries if the dataset increases.
3. Provides eventual consistency model which might lead to experiencing data loos in case of any failures

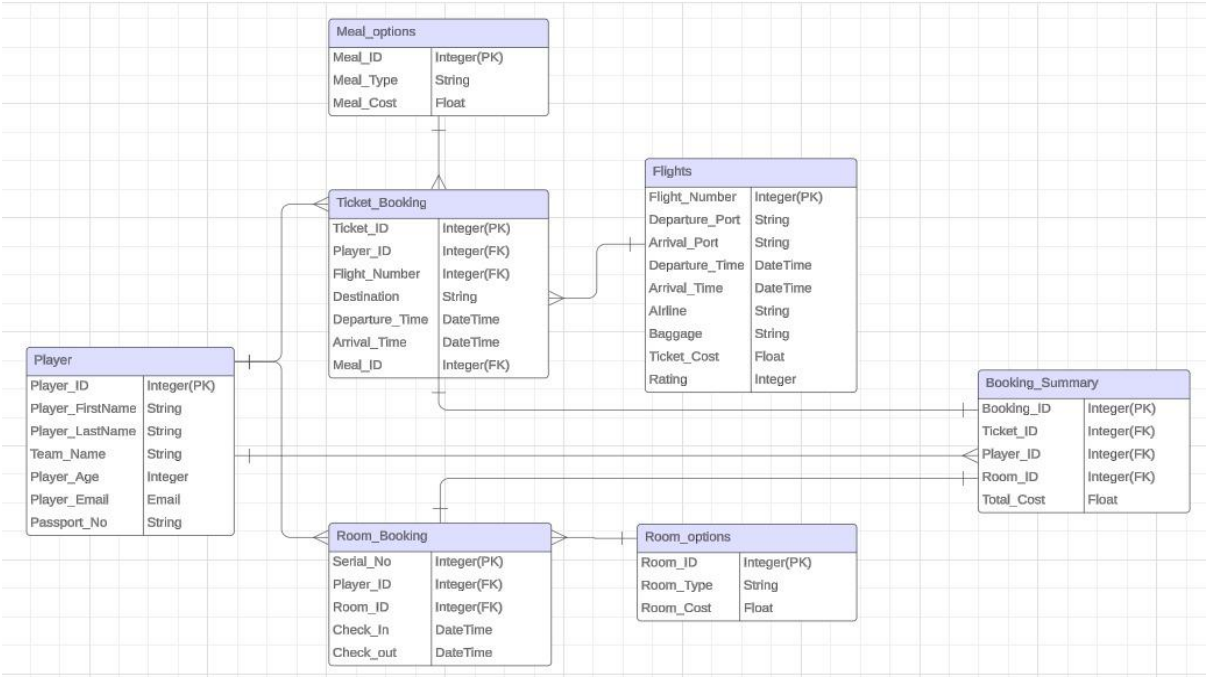
Entity-Relationship diagram: -

An Entity-Relationship Diagram (ERD) is a visual representation of the entities (such as objects, concepts, or people) within a system and the relationships between them. It depicts how these entities interact and relate to one another in a database. ERDs are useful because they provide a clear and structured way to understand the relationships between different elements in a system, aiding in database design,

communication among stakeholders, and ensuring data integrity and consistency in complex systems.

In this scenario, 7 tables have been created with entities and their relationship with other entities to implement the business case and different use case scenarios.

Figure :- Entity Relationship Diagram



These are the relationships between different tables in the database: -

1. **Player and Ticket_Booking**: A one-to-many relationship exists where a single player can have multiple ticket bookings, but each ticket booking is associated with one specific player.

2. **Player and Room_Booking**: A one-to-many relationship where one player can have several room bookings, but each room booking is linked to one particular player.

3. **Flights and Ticket_Booking**: A one-to-many relationship where one flight can be associated with multiple ticket bookings, but each ticket booking corresponds to a single flight.

4. **Meal_Options and Ticket_Booking**: A one-to-many relationship where one meal option can be chosen for multiple ticket bookings, though each ticket booking is associated with only one meal option.

5. **Room_Options and Room_Booking**: A one-to-many relationship indicating that each room type (defined in Room_Options) can be associated with many room bookings, but each room booking refers to one specific room type.

6. **Ticket_Booking and Booking_Summary**: A one-to-one relationship, assuming that each ticket booking corresponds to a single entry in the booking summary.

7. **Room_Booking and Booking_Summary**: A one-to-one relationship, assuming that each room booking is reflected once in the booking summary.

8. **Player and Booking_Summary**: A One-to-many relationship says that a player can have multiple booking summaries over a period but each summary belongs to only one player.

Logical Database Design:-

A normalization table is a structured representation of data organized into distinct tables in a relational database system. It follows the principles of database normalization, which aim to reduce redundancy and dependency within the data, thereby improving data integrity and minimizing anomalies. Each table in a normalized database represents a single entity or concept, and relationships between entities are established through keys.

1NF (First Normal Form):-

Ensures that the table has a primary key: a unique identifier for each record. - Requires that the values in each column of a table are atomic, meaning that there are no repeating groups or arrays. - Every column should contain only one value for each row (no multiple values like a list or set).

2NF (Second Normal Form):-

Meets all the requirements of 1NF. - Removes partial dependencies, which means that all non-key attributes are fully functionally dependent on the primary key. This often requires splitting up tables to ensure that each attribute is only dependent on the primary key.

3NF (Third Normal Form):-

Satisfies all conditions of 2NF. - Removes transitive dependencies. In other words, non-key attributes should not depend on other non-key attributes. This may necessitate further table splitting and reorganization to ensure every non-key attribute is only dependent on the primary key.
In this scenario, all the entities are in 3rd Normal Form and satisfies all the conditions mentioned above.

Figure:- Normal forms

1NF	2NF	3NF	Entity
Player_ID	Player_ID	Player_ID	Player
Player_FirstName	Player_FirstName		
Player_LastName	Player_LastName	Player_LastName	
Team_Name	Team_Name	Team_Name	
Player_Age	Player_Age	Player_Age	
Player_Email	Player_Email	Player_Email	
Passport_No	Passport_No	Passport_No	
Meal_ID	Meal_ID	Meal_ID	Meal_Options
Meal_Type	Meal_Type	Meal_Type	
Meal_Cost	Meal_Cost	Meal_Cost	
Ticket_ID	Ticket_ID	Ticket_ID	Ticket_Booking
Flight_Number	Flight_Number	Flight_Number	
Destination	Destination		
Departure_Time			
Arrival_Time			
Flight_Number	Flight_Number	Flight_Number	Flights
Departure_Port	Departure_Port		
Arrival_Port	Arrival_Port	Arrival_Port	
Airline	Airline	Airline	
Baggage	Baggage		
Ticket_Cost	Ticket_Cost		
Rating	Rating		
Serial_No	Serial_No	Serial_No	Room_Booking
Room_ID	Room_ID	Room_ID	
Check_In	Check_In		
Check_Out	Check_Out		
Room_ID	Room_ID	Room_ID	Room_Options
Room_Type	Room_Type	Room_Type	
Room_Cost	Room_Cost	Room_Cost	
Booking_ID	Booking_ID	Booking_ID	
Total_Cost	Total_Cost	Total_Cost	

Bracketing Notation(Database Definition Language):-

Database Definition Language (DBDL) is a notation used to describe a database schema in terms of its tables, columns, and relationships. The bracketing notation includes the table name followed by its columns in brackets, with the primary key underlined. Foreign keys are typically not underlined but are followed by an asterisk and indicate the table they reference. Let’s use the normalized structure I've provided to express this in DBDL:

This is how the normalised database structure is represented using the DBDL notation.

Player(underline{Player_ID}, Player_FirstName, Player_LastName, Team_Name, Player_Age, Player_Email, Passport_No)

Meal_Options(underline{Meal_ID}, Meal_Type, Meal_Cost)

Flights(underline{Flight_Number}, Departure_Port, Arrival_Port, Departure_Time, Arrival_Time, Airline, Baggage, Ticket_Cost, Rating)

Room_Options(underline{Room_ID}, Room_Type, Room_Cost)

Ticket_Booking(underline{Ticket_ID}, Player_ID*, Flight_Number*, Meal_ID*)

Room_Booking(underline{Serial_No}, Player_ID*, Room_ID*)

Booking_Summary(underline{Booking_ID}, Ticket_ID*, Player_ID*, Room_ID*)

In this notation:-

- underline{...} indicates the primary key of the table.
- * after a column name indicates a foreign key, with the reference to the parent table not explicitly shown in DBDL.

Since DBDL doesn't have a standard way to indicate which table a foreign key refers to, it is common practice to use naming conventions for clarity (e.g., the suffix _ID suggests a reference to an entity's primary key). When implementing this schema in a database, you would specify the exact table and column that each foreign key references.

Polyglot Persistence:-

A single database design cannot be used for all the operations in an application due to certain limitations on different types and forms of data. Polyglot persistence offers a strategic approach to database design, where different types of databases are employed within a single application. This method tailors the choice of database technology to suit the specific data storage needs of different parts of the application. By leveraging multiple database technologies, such as Relational, NoSQL, or Graph databases, developers can optimize data storage and management, leading to enhanced performance, scalability, and flexibility in the application architecture.

In a polyglot persistence architecture, each database operates independently to handle specific data or workloads, allowing them to function autonomously without requiring direct integration. For instance, in our scenario where structured data like Player profiles and particular preferences such as room, meal and flight are stored, a relational database may be utilized as it provides a schema or structure to the data. On the other hand, unstructured data like user-generated content, such as player feedback and their Special requests, may be stored in a separate NoSQL database to provide flexibility. This compartmentalized approach streamlines data access and manipulation, facilitating efficient application logic implementation.

Relational databases are not always suitable for special requests and feedback data due to their rigid schema requirements, complex relationship representations, and limitations in handling unstructured or semi-structured data efficiently. Additionally, relational databases may face scalability and performance issues when dealing with large volumes of data or real-time processing requirements, whereas NoSQL databases offer more flexibility and scalability for such use cases.

MongoDB emerges as a superior option compared to Redis, which is a Key-Value based database, or Graph structures for storing player feedback and special requests due to its document-oriented storage model. MongoDB's document-oriented approach accommodates complex and hierarchical data structures, making it well-suited for representing diverse types of player feedback and special requests made by them. Each feedback or request can be stored as a document,

encompassing various attributes such as customer details, timestamps, comments, ratings, and specific requests. This inherent flexibility enables MongoDB to adapt seamlessly to changing requirements over time, without necessitating a predefined schema.

CRUD Operations :-

Read:-

Player Profiles: The read operation involves accessing player profiles stored in the database. These profiles contain comprehensive information about each player, including personal details and any other relevant information.

Flight Information: Accessing flight details from the database allows the application to retrieve comprehensive information about available flights.
It collects essential details such as departure and arrival dates, preferred airlines, and meal preferences.

```
# Selecting a particular Airline as they might be a loyal customers.
queryCase2 = '''
SELECT TOP 1 * FROM AirBuddy.Flights
WHERE Airline = 'British Airways'AND
      CAST(Departure_Time AS DATE) = '2024-04-01'AND
      Departure_Port = 'London'AND Arrival_Port = 'Mumbai'
      ORDER BY DATEDIFF(minute, Departure_Time, Arrival_Time);
'''
```

```
queryToFetchRequiredPlayers = '''
SELECT Player_ID
FROM AirBuddy.Players
WHERE Team_Name = 'England'
AND (Player_FirstName = 'Ben' AND Player_LastName = 'Stokes'
     OR Player_FirstName = 'Jason' AND Player_LastName = 'Roy'
     OR Player_FirstName = 'Ben' AND Player_LastName = 'Foakes'
     OR Player_FirstName = 'Ollie' AND Player_LastName = 'Pope'
     OR Player_FirstName = 'Roy' AND Player_LastName = 'Burns'
     OR Player_FirstName = 'Sam' AND Player_LastName = 'Curran'
     OR Player_FirstName = 'Tom' AND Player_LastName = 'Curran'
     OR Player_FirstName = 'Jack' AND Player_LastName = 'Leach'
     OR Player_FirstName = 'Craig' AND Player_LastName = 'Overton'
     OR Player_FirstName = 'James' AND Player_LastName = 'Anderson'
     OR Player_FirstName = 'Jofra' AND Player_LastName = 'Archer'
     OR Player_FirstName = 'Zak' AND Player_LastName = 'Crawley'
     OR Player_FirstName = 'Jonny' AND Player_LastName = 'Jonny Baristow'
     OR Player_FirstName = 'Joe' AND Player_LastName = 'Joe Root'
     OR Player_FirstName = 'Sam' AND Player_LastName = 'Billings');
'''
```

Create:-

Flight Reservation: Based on the gathered information, the application proceeds to make reservations with the chosen airlines for the entire team. This involves securing appropriate flights that align with the team's schedule and preferences. This creates new records for the bookings done for the England players travelling to Mumbai from London.

```
def generate_unique_id():

    return '{:04}'.format(random.randint(2050, 9999))
flight_number = 200001
departure_time = '2024-04-01 08:30:00'
arrival_time = '2024-04-02 14:30:00'
player_ids = [46, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60]

# Execute the INSERT statement for each player ID
for player_id in player_ids:
    # Generate a unique 4-digit random ID
    ticket_id = generate_unique_id()

    # Execute the INSERT statement
    cursor.execute('''
        INSERT INTO AirBuddy.TicketBookings (Ticket_ID, Player_ID, Flight_Number, Departure_Time, Arrival_Time, Meal_ID)
        VALUES (?, ?, ?, ?, ?, ?)
    ''', (ticket_id, player_id, flight_number, departure_time, arrival_time, random.randint(1, 10)))
cnxn.commit()
```

Update:-

Meal Preferences: The update operation facilitates modifications to existing data in the database. For this case, it enables updating meal preferences for players from their previously opted choices to Vegetarian meals during the flight journey.

```
# Preferring only vegetarian meal during their travel.
mealUpdateQuery ='''
UPDATE AirBuddy.TicketBookings
SET Meal_ID = (
    SELECT Meal_ID
    FROM AirBuddy.MealOptions
    WHERE Meal_Type = 'Vegetarian'
)
WHERE Player_ID IN (55, 60)
AND Flight_Number = 200001;
'''
```

Delete:-

The delete operation involves removing records from the database. In this situation when a player drops out of the tour due to injury or any other reason, their information, including flight bookings and preferences, needs to be deleted from the summary table.

Additionally, if a new player joins the team as a replacement, their information will be added to the database, including flight bookings and any specific preferences or requirements.

```
# Delete the existing record for player 49
deleteRecordquery = '''
DELETE FROM AirBuddy.TicketBookings
WHERE Player_ID = 49
AND Flight_Number = 200001;'''

cursor.execute(deleteRecordquery)

new_ticket_id = generate_unique_id()
# -- Insert a new record for player 41
cursor.execute('''
        INSERT INTO AirBuddy.TicketBookings (Ticket_ID, Player_ID, Flight_Number, Departure_Time, Arrival_Time, Meal_ID)
        VALUES (2049, 41, 200001, '2024-04-01 08:30:00.000', '2024-04-02 14:30:00.000', 3)
    ''', )
cnxn.commit()
```

With keeping Novelty in mind, we have summarized the data to create a column in the summary to include total cost to be paid by player based on his preferences in all sections. The calculations are done by taking cost values from different tables which are not related to each other. Here is a code snippet and the output for the same showing in the final column the cost debt of each player.

```
query = '''
SELECT BS.Booking_ID,
       P.Player_FirstName,
       P.Player_LastName,
       F.Ticket_Cost AS Ticket_Cost,
       MO.Meal_Cost AS Meal_Cost,
       RO.Room_Cost AS Room_Cost,
       DATEDIFF(day, RB.Check_IN, RB.Check_OUT) AS Days_Spent,
       (F.Ticket_Cost + MO.Meal_Cost + (DATEDIFF(day, RB.Check_IN, RB.Check_OUT) * RO.Room_Cost)) AS Total_Cost
FROM AirBuddy.BookingSummary BS
JOIN AirBuddy.Players P ON BS.Player_ID = P.Player_ID
JOIN AirBuddy.TicketBookings TB ON BS.Ticket_ID = TB.Ticket_ID
JOIN AirBuddy.Flights F ON TB.Flight_Number = F.Flight_Number
JOIN AirBuddy.MealOptions MO ON TB.Meal_ID = MO.Meal_ID
JOIN AirBuddy.RoomBookings RB ON BS.Room_ID = RB.Serial_No
JOIN AirBuddy.RoomOptions RO ON RB.Room_ID = RO.Room_ID;
'''
```

```
(110, 'Sarfaraz', 'Khan', 620.0, 12.75, 48.0, 3, 776.75)
(111, 'Shubham', 'Gill', 620.0, 10.5, 90.0, 3, 900.5)
(112, 'Rohit', 'Sharma', 620.0, 11.25, 80.0, 3, 871.25)
(113, 'Dhruv', 'Jurel', 620.0, 11.75, 60.0, 3, 811.75)
(114, 'Virat', 'Kohli', 620.0, 14.5, 75.0, 3, 859.5)
(115, 'Sarfaraz', 'Khan', 450.0, 12.0, 48.0, 3, 606.0)
(116, 'Rohit', 'Sharma', 450.0, 15.25, 80.0, 3, 705.25)
(117, 'Dhruv', 'Jurel', 450.0, 11.25, 60.0, 3, 641.25)
(118, 'Shubham', 'Gill', 450.0, 14.5, 90.0, 3, 734.5)
(119, 'Rishabh', 'Pant', 450.0, 12.75, 40.0, 3, 582.75)
(120, 'Shreyas', 'Iyer', 450.0, 11.75, 65.0, 3, 656.75)
(121, 'KL', 'Rahul', 450.0, 13.0, 55.0, 3, 628.0)
(122, 'KS', 'Bharath', 450.0, 9.75, 90.0, 3, 729.75)
(123, 'Virat', 'Kohli', 450.0, 13.5, 75.0, 3, 688.5)
(124, 'Yashasvi', 'Jaiswal', 450.0, 10.5, 35.0, 3, 565.5)
```

Conclusion:-

Through the strategic implementation of both relational and NoSQL databases, Air Buddy ensures not only data integrity and scalability but also adaptability to the diverse and dynamic nature of sports travel. The polyglot persistence approach underscores its commitment to flexibility and performance, allowing for seamless management of both structured and unstructured data.

The entity-relationship diagram (ERD) serves as a visual testament to the meticulous database design, where every relationship is carefully crafted to optimize data management and retrieval. The adherence to normalization principles not only enhances efficiency but also fortifies the system against potential anomalies and inconsistencies.

In harnessing the power of CRUD operations, Air Buddy empowers sports teams to navigate through every facet of their travel journey with precision and ease. From creating personalized itineraries to seamlessly updating preferences and handling unforeseen changes, Air Buddy stands as a beacon of reliability and efficiency.

References: -

- ESPN Cricinfo. (n.d.). Cricketers. Retrieved from https://www.espncricinfo.com/cricketers

- Conde Nast Traveler. (2016, July 8). British Airways flight turns around after almost 12 hours in air. Retrieved from https://www.cntraveler.com/stories/2016-07-08/british-airways-flight-turns-around-after-almost-12-hours-in-air

- TechTarget. (n.d.). Polyglot persistence. In SearchAppArchitecture. Retrieved from https://www.techtarget.com/searchapparchitecture/definition/polyglot-persistence

- MongoDB. (n.d.). Document databases. Retrieved from https://www.mongodb.com/document-databases