Travel Booking System

Application Suitability: Instead of building a monolithic application, this system is composed of multiple independent microservices, each responsible for a specific function or feature. These microservices can include components for user authentication, search and reservation, and payment processing.

As a real life example of my idea is TravelPerk, which is an all-in-one digital platform for businesses and corporate travelers. It empowers travelers to handle their entire travel booking process independently, providing access to an unrivaled travel inventory for flights, accommodation, train travel, and rental cars.

Service Boundaries:

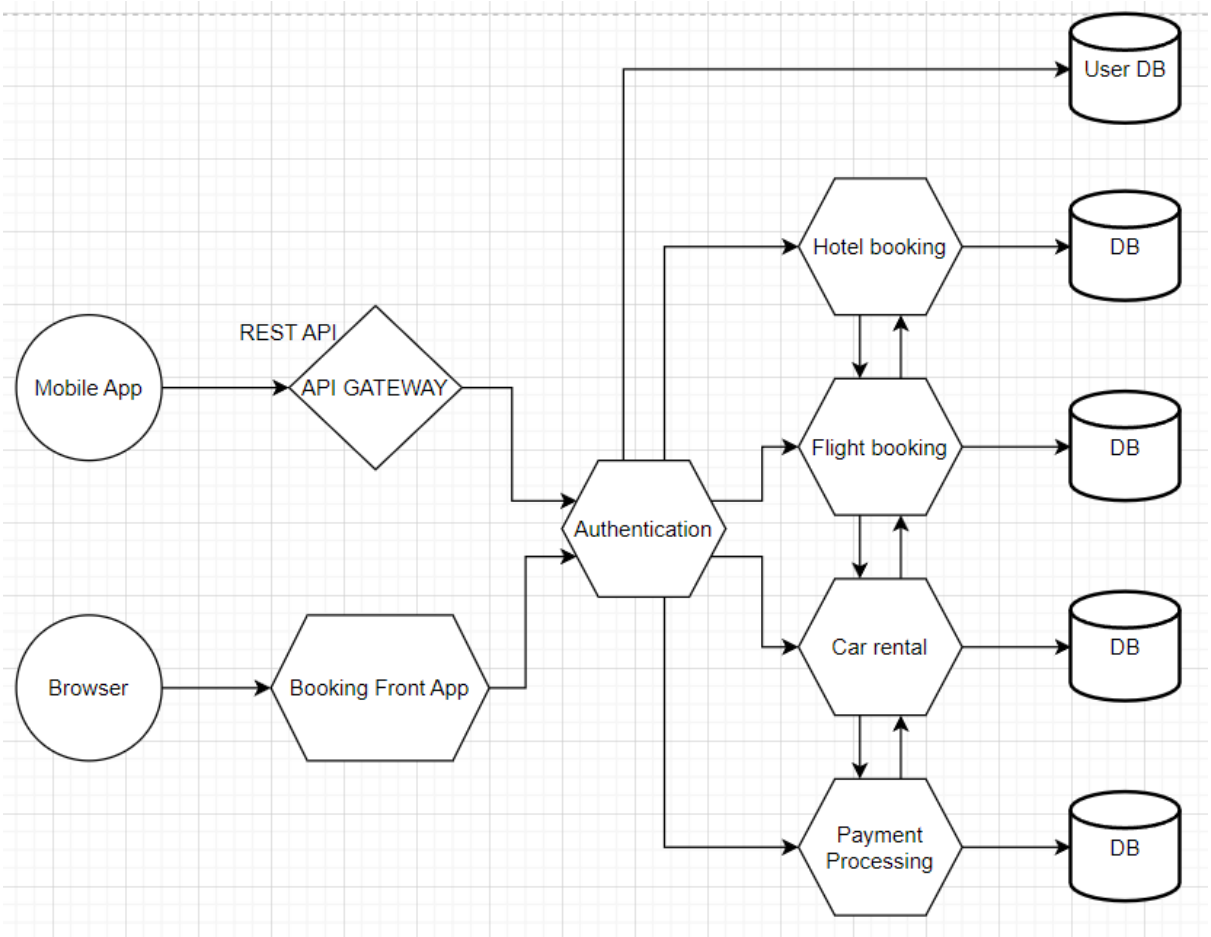

Fig. 1. Travel Booking System Architecture

Technology Stack and Communication Patterns: Here's a overview of how the structure of my Travel Booking System works, tools and communication patterns that I will be using in both languages Python and C#:

- Backend Server (Python):

Using Python to create the backend server that handles the core logic of your Travel Booking System, web framework for python Flask.

- Database (Python and SQL):

Using Python to interact with the database, where I will store information about flights, hotels, customers, reservations, etc.

- Frontend (C#):

Using C# with technologies like Windows Forms or WPF building a desktop application.

- Communication:

Using APIs (Application Programming Interfaces) to establish communication between my Python backend and C# frontend. RESTful APIs is my choice, I will create APIs using Python's Flask REST framework and consume them in my C# application.

- Payment Processing:

Using Python or C# libraries that provide payment gateway integration (Stripe from Python or PayPal from C#).

- User Authentication:

Using libraries like Flask-Login built-in authentication system will handle user authentication and authorization on the backend using Python.

Design Data Management: Will be using 5 separate DataBases, one for the Hotel reservations, one for the booking Fights, one for renting Cars, one for the payments and the last one for user storage. Once the user booked a hotel, a flight or a car, the id of the user and of the item booked depending on the type will go to the database. User will have 30 minutes to proceed with payment, if he doesn't the items will be back on the site, if he does the payment id and the user id will go to Payment DataBase.

JSON example for the Hotel Reservation:


```
{
  "reservation_id": "123456789",
  "customer_name": "John Smith",
  "check_in_date": "2023-10-15",
  "check_out_date": "2023-10-20",
  "hotel_name": "Cristal Hotel",
  "total_price": 750.00,
  "reservation_status": "Confirmed"
}
```


JSON example for the Flight Booking:


```
{
  "reservation_id": "ABC123",
  "customer_name": "John Smith",
  "flight_number": "AA123",
```

```
  "departure_date": "2023-11-15",
  "departure_airport": "JFK",
  "destination_airport": "LAX",
  "seat_number": "23A",
  "ticket_price": 350.00,
  "reservation_status": "Confirmed"
}
```


JSON example for the Car rental:


```
{
  "reservation_id": "C12345",
  "customer_name": "John Smith",
  "rental_start_date": "2023-10-15",
  "rental_end_date": "2023-10-20",
  "car_type": "SUV",
   "total_cost": 450.00,
  "reservation_status": "Confirmed"
}
```

JSON example for the payment:


```
{
  "transaction_id": "123456789",
  "payment_date": "2023-10-25T14:30:00Z",
  "payer_name": "John Smith",
  "payment_method": "Credit Card",
  "amount": 1450.00,
  "currency": "USD",
  "invoice_number": "INV-2023-001",
  "description": "Hotel Reservation Payment, Flight Booking Payment, Car rental Payment",
  "status": "Success"
}
```


JSON example of user authentication:


```
{
  "user_id": "123456",
  "username": "John Smith",
  "email": "johnsmith@utm.com",
  "password_hash": "hashed_password"
}
```

Deployment and Scaling:

Will be using Docker, because it encapsulates an application and its dependencies, including libraries and runtime environments, it ensures consistency between development, testing, and production environments. Also, Docker containers can be started and stopped quickly, making them ideal for microservices and applications that need to scale up or down rapidly in response to changing workloads.