Software Engineering Project 3

Team 13 - Gaurav, Kabir, Mahika, Vanshita, Vyom

Task 1

Functional Requirements:

1. **User Registration:**
    - Users should be able to create new accounts by providing their name, email, phone number, and password.
    - The system should validate user-provided information and store it securely.
2. **User Login:**
    - Registered users should be able to log in using their email and password.
    - The system should verify login credentials through secure hash checks.
    - Architecturally, it necessitates the implementation of secure hash algorithms and protocols for verifying user credentials. The system's authentication module must be robust to prevent unauthorized access and protect user accounts from potential security threats.
3. **Hotel Booking:**
    - Users should be able to search for hotels by specifying location, distance from city center, rating and any services provided by the hotel.
    - The system should display available hotels and allow users to book rooms by paying the calculated fare using options for payment via UPI or credit card.
    - Upon successful booking, the system should confirm the reservation and redirect users to the My Bookings page.
4. **Travel Itinerary Generation:**
    - Users should be able to input their travel details such as destination city, start and end dates, interests, and additional notes.

- The system should generate a personalized day-wise itinerary based on user preferences, suggesting activities, attractions, and dining options for each day of the trip.
- Architecturally, it requires designing interfaces for collecting user preferences, integrating with external APIs for data retrieval, itinerary generation for performance and relevance.

5. **Travel Partner Matching:**
    - Users should be able to search for travel partners by specifying location, city, month, and interests.
    - The system should match users with compatible travel partners and provide contact options for collaboration.
6. **Weather Alert Subscription:**
    - Users should be able to subscribe to weather alerts for specified locations.
    - The system should dynamically fetch weather information every hour and send alerts to subscribed users.
    - Architecturally, it necessitates designing event-driven architectures for fetching and processing weather data, implementing subscription management systems, and integrating with notification services for timely delivery.
7. **Payment Processing:**
    - Users should be able to make payments for hotel bookings using UPI or credit card.
    - The system should securely handle payment transactions and provide confirmation of successful bookings.

Non-functional Requirements:

1. **Performance:**
    - The system should respond to user interactions promptly, especially during peak usage times.
    - For example, the Hotel Booking Subsystem needs to calculate fares and process payments quickly to prevent user frustration and ensure a smooth booking experience.
2. **Security:**

- All subsystems should adhere to stringent security measures to protect user data and transactions.
- For instance, the User Login and Registration Subsystem must employ encryption techniques to safeguard sensitive information like passwords and personal details from unauthorized access or data breaches. Also, only the email and phone number of the users are getting displayed with matching interests in the Travel Partner Feature.

3. **Scalability:**
   - As user traffic increases over time, the system should be able to handle the growing load without compromising performance.
   - For example, the Weather Alert Subsystem needs to dynamically scale its resources to accommodate fluctuating demand for real-time weather updates, ensuring uninterrupted service during peak periods.
4. **Reliability:**
   - **Explanation:** The system should operate reliably without frequent downtime or disruptions. For instance, the Travel Partner Subsystem needs to reliably match users with compatible travel partners and deliver notifications without errors to maintain user trust and satisfaction.
5. **Availability:**
   - All subsystems should be available for use whenever users require them.
   - For example, the Payment Subsystem must be available 24/7 to facilitate hotel booking payments, ensuring users can complete transactions at their convenience without encountering downtime.
6. **Usability:**
   - The user interface of each subsystem should be intuitive and easy to navigate, catering to users of varying technical abilities.
   - For instance, the Travel Itinerary Subsystem should present the day-wise itinerary in a clear and organized manner, allowing users to understand and customize their travel plans effortlessly.
7. **Maintainability:**

- The system should be designed and implemented in a way that facilitates easy maintenance and updates.
- All subsystems follow modular and well-documented code practices, allowing developers to make changes or enhancements efficiently without disrupting the overall functionality of the system.

Subsystem Overview:

1. **User Login and Registration**
    - **Role:** Manages user authentication through login and registration processes, ensuring secure access to the trip planner web app.
    - **Functionality:** The User Login and Registration Subsystem facilitates user authentication within the trip planner web app. For new users, the system collects essential information including name, email, phone number, and password during the registration process. User passwords are securely hashed before storage. For existing users, the system prompts for their registered email and password during login, performing a hash check to verify credentials. This subsystem ensures the integrity and security of user accounts, enabling seamless access to personalized features and services.
2. **Trip Itinerary**
    - **Role:** The travel itinerary subsystem, a key component of the trip planner web app, acts as the user's virtual travel assistant. It interprets user inputs, including destination details and preferences, to craft a bespoke itinerary spanning from the start to the end date of the trip.
    - **Functionality:** The travel itinerary subsystem of the trip planner web app facilitates efficient trip planning for users. It collects user inputs such as destination city, start and end dates, interests, and notes. The system then processes this data to generate a personalized day-wise itinerary, suggesting activities, attractions, and restaurants tailored to the user's preferences for each day of the trip, from the start day to the departure day using the LLM Gemini. It allows for itinerary customization and provides recommendations, presenting the finalized plan in a user-friendly format.
3. **Travel Partner**

- **Role:** The Travel Partner Subsystem within the trip planner web app efficiently connects users with compatible travel partners, facilitating collaboration and enhancing the overall travel planning experience**.**
- **Functionality:** The Travel Partner Subsystem within the trip planner web app facilitates users in finding compatible travel partners by inputting desired location, city, month, and interests. It searches for potential partners with similar preferences, providing a list for user review and offering contact options through the platform, including phone numbers and emails, to foster collaboration.

3. **Hotel Search**

- **Role:** The Hotel Booking Service subsystem of the trip planner web app acts as a seamless interface, to fetch and present a list of available hotels based on user preferences, facilitating direct booking and enhancing the overall travel planning experience.
- **Functionality**: The Hotel Booking Service subsystem of the trip planner web app streamlines the hotel search and booking process for users. Through a user-friendly interface, users provide input such as city name, distance from the city center, desired amenities (such as swimming pool, spa, fitness center, restaurant, air conditioning), and minimum rating. Leveraging this information, the system conducts a tailored search, utilizing the Amadeus API to fetch and present a list of available hotels meeting the specified criteria. Users are then offered the option to directly book their chosen hotel through the platform.

4. **Hotel Booking**

- **Role:** The Hotel Booking Subsystem within the trip planner web app efficiently processes hotel bookings, handling fare calculation and payment processing, ultimately confirming bookings and collecting feedback for user improvement.
- **Functionality:** The Hotel Booking Subsystem within the trip planner web app facilitates seamless hotel bookings for users. Users input their check-in and check-out dates, along with the number of guests. The system calculates the fare for the stay and offers two buttons: *"Calculate Fare"* and *"Proceed to Payment"*. Upon clicking "Calculate Fare," the fare is computed based on number of days of stay, number of guests and rating of the hotel, and clicking "Proceed to Payment" initiates the payment process. Once payment is completed, the system confirms the booking and collects user feedback for continuous improvement.

5. **Payment:**
    - **Role:** Facilitates secure payment processing for hotel bookings, offering options for UPI or credit card payments.
    - **Functionality:** The Payment Subsystem seamlessly handles payment transactions for hotel bookings initiated from the Hotel Booking page. Users are redirected here and presented with two payment options: UPI or credit card. If the UPI option is chosen, the system prompts users to input their UPI ID and UPI PIN, allowing them to confirm or cancel the booking. Alternatively, if paying by credit card, users provide their credit card number and CVV. Once payment is completed, the system redirects users to the My Bookings page, providing a smooth and secure transaction experience.
6. **Weather Alerts:**
    - **Role:** Facilitates weather alerts for subscribed users based on their specified location using the MeteoSource API.
    - **Functionality:** The Weather Alert Subsystem within the trip planner web app enables users to receive real-time weather updates for their specified locations. Users provide their desired city and subscribe to weather alerts, allowing the system to dynamically fetch weather information every hour using the MeteoSource API. The system delivers a summary of weather conditions, temperature, wind speed, and precipitation details to subscribed users. Additionally, users have the option to unsubscribe from weather alerts whenever they wish, providing flexibility in managing their notifications

Task 2

In this task, the stakeholders, their concerns, and the viewpoints and views addressing these concerns have been identified according to the IEEE 42010 standard.

Task 2a

Stakeholders

- travellers
- payment gateway service
- weather prediction service

- transport services
- accommodation services
- destination management organization (DMOs)
- administrator
- software developers

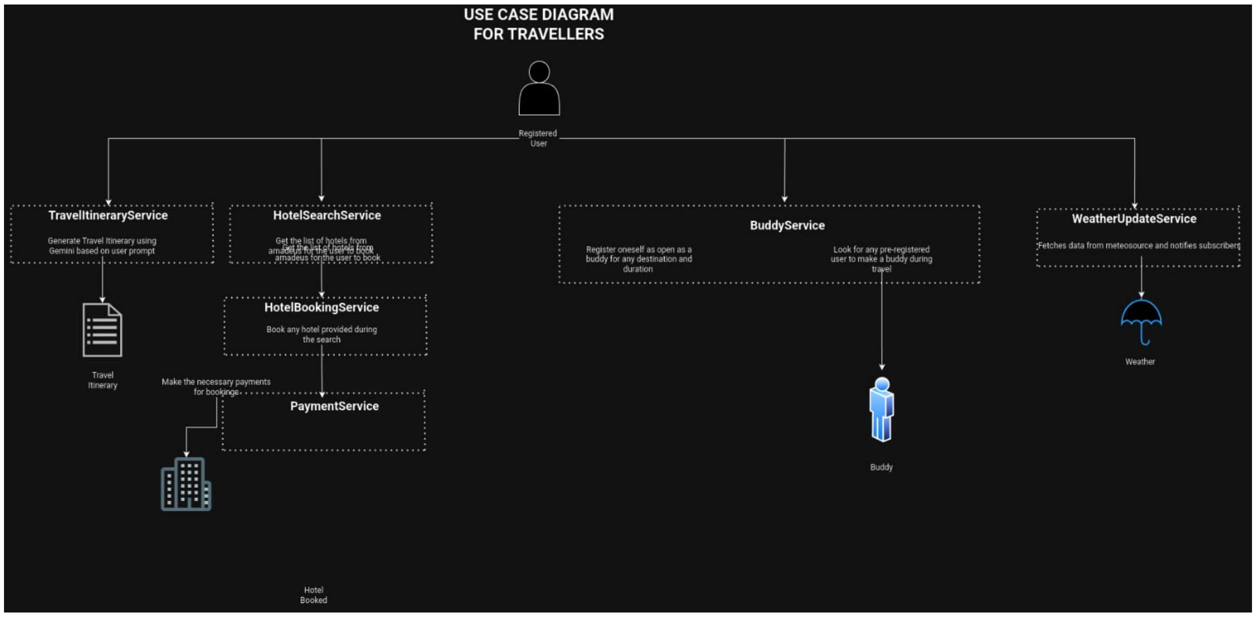
Concerns

For each stakeholder, the concerns have been mentioned below:

1. **Travellers** - Travellers are concerned with being given a one stop solution to all planning behind a trip/vacation comprising generation of itinerary, making travel and accommodation bookings, booking tickets for destinations as well as finding a suitable travel buddy. An easy to use interface and end-to-end functionality from a single app is what is being demanded by the user.
2. **Payment Gateway Service** - A payment gateway is being applied to get payments for orders made for transport and bookings. The only concern for the gateway is suitable incorporation to the existing architecture and feasibility to handle payments made.
3. **Weather Prediction Service** - The weather prediction service is needed to alert and notify the users about the weather conditions before or during the travel. Accurate prediction on time along with proper notification is suitable and required for the system.
4. **Transport, Accommodation Services & DMOs** - Any transport, accommodation as well as DMO agency willing to provide service for any mode wants it’s service to come under scrutiny when demanded as well as be provided adequate payment for any provided service. For suitable functioning, all transport timings and accommodation details must be visible during bookings and payment as well as trip records should be made. Proper interface should be given to register themselves and give details of the services on the app.
5. **Administrator** - The administrator is responsible for adding, verifying users as well as any sort of service providers to the app. They would also have overall access to all information of the app.
6. **Software Developers** - Software developers are concerned to make the application highly modular and extensible to incorporate addition of new features in the future. Making use of correct patterns in terms of design and architecture are important.

Viewpoint & Views

For travelers, the use case diagram has been given below which also covers stakes of payment and hotel services.


USE CASE DIAGRAM FOR TRAVELLERS
Registered User
TravelItineraryService
Generate Travel Itinerary using Gemini based on user prompt
Travel Itinerary
HotelSearchService
Get the list of hotels from amadeus for the user to book
HotelBookingService
Book any hotel provided during the search
Make the necessary payments for bookings
PaymentService
Hotel Booked
BuddyService
Register oneself as open as a buddy for any destination and duration
Look for any pre-registered user to make a buddy during travel
Buddy
WeatherUpdateService
Fetches data from meteosource and notifies subscribers
Weather

Task 2b

Major Design Decisions and Rationale

Choosing Architecture Pattern

Context

Choosing an architecture pattern to simplify the codebase, as well making it more extensible, modular, understandable and scalable is a desired task.

Decision

The chosen decision is to use **Microservices** because we have a set of independent and limited functionalities. Thus, using microservices is useful because we can create separate channels for each functional and non-functional requirement ensuring that they do not interfere with each other and we also prevent cascading failures. Further, the codebase remains highly scalable, as each service can be modified as needed. It also isolates any sort of faults and makes the programming much easier.

Choosing Design Pattern for Payments

Context

Implementing multiple payment methods is a tedious task. Also, it is desired to ensure that one can extend and add multiple payments methods to the existing codebase and edit the functioning of one.

Decision

The chosen decision is to use strategy design pattern. Implementing separate strategies for separate payment methods is done. This provides us with an additional benefit of making one capable of adding more and more payment methods as desired. Further, it reduces the conditional complexity of choosing the payment method and dedicates common functions to make the payments.

Choosing LLM for Itinerary Generation

Context

To generate a customized and personalised itinerary, some automated system is required in the form of some LLM for optimal results.

Decision

Gemini had been chosen for this task because the APIs for the same are easily accessible, free as well as have higher upper limits. Thus, it is quite suitable for our task. Additionally we had used it before for some previous tasks making it much easier to implement.

Choosing Database

Context

An efficient database is needed to store user details, hotel details, preferences, payment methods, etc. which stores them separately and makes the data easily and quickly accessible.

Decision

We chose MySQL for the above task. MySQL is a relational database unlike MongoDB. This makes storing much more organised and retrieval more understandable as well as quicker. It is easier to implement and is stored over the local system unlike on some cloud feature making it quite suitable.

Task 3

Architectural Tactics Used

1. **Exception Handling for Improved Availability:**
Exception handling is a critical architectural tactic employed to bolster the non-functional requirement of "Availability" within the system. When backend API calls encounter faults or errors, the system gracefully manages these exceptions, preventing a complete shutdown. Instead, it responds with informative error messages, ensuring that the system remains operational and responsive to user requests. By proactively addressing potential failures, this tactic enhances the system's overall reliability and availability, thereby promoting uninterrupted service delivery.

2.
Semantic Coherence for Enhanced Modifiability:
Semantic coherence is meticulously maintained throughout the codebase, serving as a fundamental tactic to facilitate "Modifiability" within the system. By ensuring that the code's semantics remain consistent and comprehensible across all components, developers can seamlessly add, modify, or extend functionality without introducing unnecessary complexity or ambiguity. This cohesive approach to code organization not only simplifies the development process but also fosters a more maintainable and adaptable architecture, enabling efficient evolution in response to changing requirements and business needs.

3.
Cancellation of Transactions and UI Separation for Improved Usability:
Two key tactics have been implemented to enhance the "Usability" of the system: the ability to cancel payment transactions midway and the separation of the user interface (UI) from other system files. By integrating features that allow users to cancel transactions seamlessly, the system promotes a positive user experience by providing flexibility and control over their

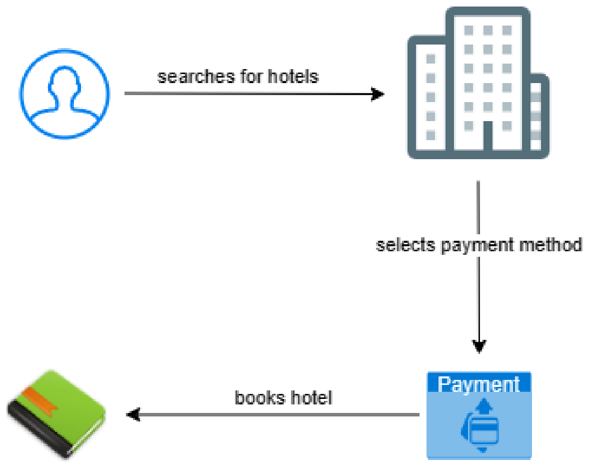
interactions. Additionally, isolating the UI from the core system logic improves usability by creating an intuitive and user-friendly interface that remains independent of backend operations. This design choice not only simplifies navigation and interaction but also enhances the overall accessibility and usability of the system for a diverse range of users.

4.
Password Hashing and Data Integrity for Enhanced Security:
Robust security measures, including password hashing and data integrity checks, are integral components of the system architecture, bolstering its resilience against potential threats and vulnerabilities. User passwords stored in the database are securely hashed, ensuring that sensitive information remains protected even in the event of a security breach. Furthermore, stringent data integrity checks are enforced throughout the system, verifying the accuracy and consistency of user inputs to maintain a reliable and trustworthy data environment. By prioritizing security at every level of the architecture, the system safeguards confidential information, preserves the integrity of user data, and instills confidence in its users, thereby fulfilling the critical non-functional requirement of "Security".

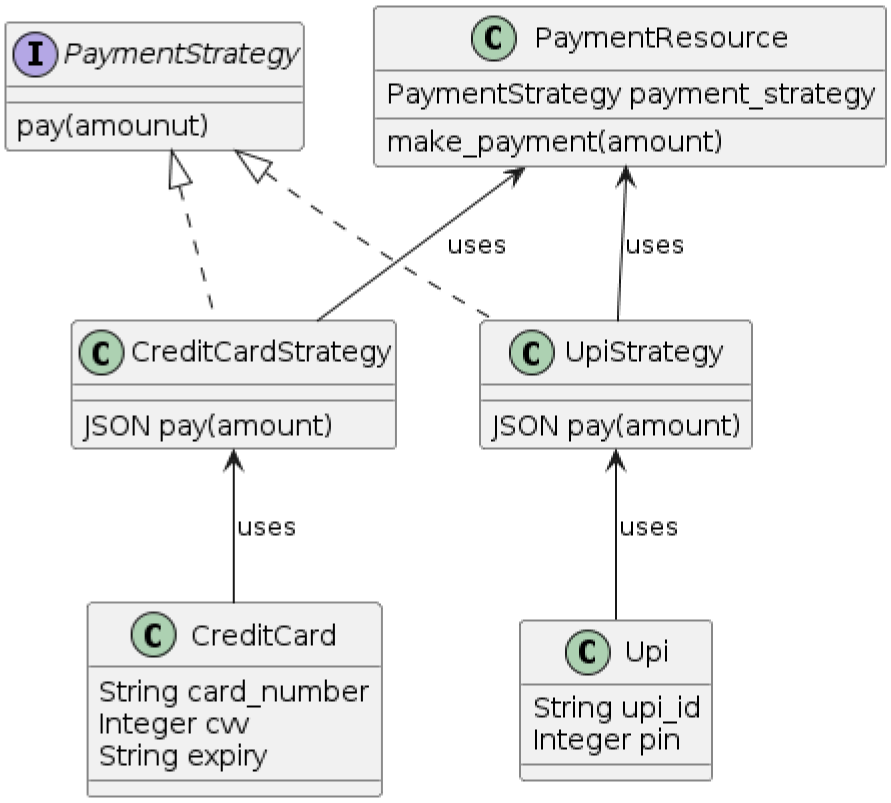
Design Patterns

Strategy Pattern

Strategy Pattern has been used to implement the payment service of the app. For different payment methods, separate strategies have been implemented. This makes it much easier to extend the codebase further to multiple methods as well manage bugs and existing methods with ease.

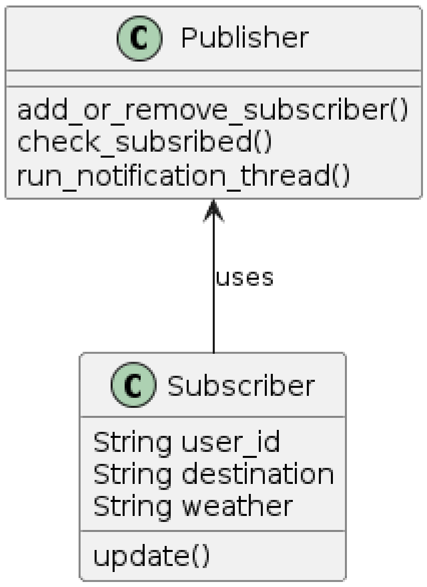


```
## PaymentStrategy
from abc import ABC,abstractmethod
class PaymentStrategy(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

## CreditCardStrategy
from .PaymentStrategy import PaymentStrategy
from flask import jsonify
from app import db
from app.models.CreditCard import CreditCard
import datetime

class CreditCardStrategy(PaymentStrategy):
    def __init__(self, data):
        self.data = data

    def pay(self, amount):
        # Assuming 'data' contains credit card credentials
        credit_card_number = self.data.get('cardNumber')
        cvv = self.data.get('cvv')
        expiry_date = self.data.get('expiryDate')
        expiry_date_obj = datetime.datetime.strptime(expiry_d
ate, '%Y-%m')
        expiry_date_formatted = expiry_date_obj.strftime('%m
%Y')
        print(credit_card_number)
        print(cvv)
        print(expiry_date_formatted)
        try:
            credit_card = CreditCard.query.filter_by(card_num
ber=credit_card_number,
                                                         cvv=cvv,
                                                         expiry=e
xpiry_date_formatted).first()

            if credit_card:
                response = {'message': f"Paid {amount} from C
ard","status":200}
            else:
                response = {'message': 'Incorrect credit card
credentials',"status":400}

        except Exception as e:
            print("Error:", e)
            response = {'message': 'An error occurred while p
rocessing the payment',"status":400}

        # Convert the response to JSON and return
        return jsonify(response)

## UpiStrategy
from .PaymentStrategy import PaymentStrategy
from app.models.Upi import UPI
from flask import jsonify

class UpiStrategy(PaymentStrategy):
    def __init__(self, data):
        self.data = data

    def pay(self, amount):

        upi_id = self.data.get('upiId')
        pin = self.data.get('upiPin')
        upi_details = UPI.query.filter_by(upi_id=upi_id, pin=
pin).first()

        if upi_details:
            res =  f"Paid {amount} via UPI"
            return jsonify({"message":res,"status":200})
        else:
            res =  "UPI payment failed: Incorrect UPI ID or P
IN"
            return jsonify({"message":res,"status":400})

## PaymentResource
from .CreditCardStrategy import CreditCardStrategy
from .UpiStrategy import UpiStrategy
class PaymentResource:
    def __init__(self, payment_method,data):
        if(payment_method=="Card"):
            self.payment_strategy = CreditCardStrategy(data)
        elif(payment_method=="UPI"):
            self.payment_strategy = UpiStrategy(data)

    def make_payment(self, amount):
        return self.payment_strategy.pay(amount)
```



Observer Pattern

For the weather updates, users have to be notified if they subscribe for the same at some particular location. Thus, observer pattern is the most suitable for the above task. It becomes much easier to not only subscribe to multiple locations but also making notifications in the backend is feasible.



Implementation of the same is being given below in detail:

```
## SubscriberModel.py
from app import db
import requests


class Subscriber(db.Model):
    __tablename__ = "subscribers"

    user_id = db.Column(db.String(255), primary_key=True)
    destination = db.Column(db.String(255), nullable=False)
    weather = db.Column(db.String(1000))

    def update(self):
        try:
            response = requests.get(
                f"https://www.meteosource.com/api/v1/free/poi
nt?place_id={self.destination}&sections=current&timezone=auto
&language=en&units=auto&key=6aui4vgiqcq9wfgbmit1u1357n6f3si08
je5mus3"
            )
            if response.ok:
                data = response.json()
                print(data.get("current"))
                current_weather = data.get("current")
                if current_weather:
                    self.weather = current_weather
                    db.session.commit()
            else:
                print(
                    f"Failed to fetch weather data for subscr
iber {self.user_id}. Status code: {response.status_code}"
                )
        except Exception as e:
            print(
                f"Error updating weather data for subscriber
{self.user_id}: {str(e)}"
            )

## Publisher.py
from flask import request, jsonify, session
from app import app, db
from app.models.SubscriberModel import Subscriber
from flask import request
import time
import threading


@app.route("/add_or_remove_subscriber", methods=["POST"])
def add_or_remove_subscriber():
    print(request.form)
    user_id = request.form.get("user_id")
    destination = request.form.get("destination")

    existing_subscriber = Subscriber.query.filter_by(user_id=
user_id).first()

    if existing_subscriber:
        db.session.delete(existing_subscriber)
        db.session.commit()
        return (
            jsonify(
                {"message": f"Subscriber with user_id {user_i
d} removed successfully"}
            ),
            200,
        )
    else:
        if not user_id or not destination:
            return jsonify({"error": "Missing user_id or dest
ination"}), 400
        new_subscriber = Subscriber(
            user_id=user_id, destination=destination, weather
=""
        )
        db.session.add(new_subscriber)
        db.session.commit()
        new_subscriber.update()
        return (
            jsonify(
                {"message": f"Subscriber with user_id {user_i
d} added successfully"}
            ),
            200,
        )
```

```
@app.route("/check_subscribed/<user_id>", methods=["GET"])
def check_subscribed(user_id):
    subscriber = Subscriber.query.filter_by(user_id=user_id).
first()
    if subscriber:
        return jsonify({"subscribed": True,"weather":subscrib
er.weather,"location":subscriber.destination}), 200
    else:
        return jsonify({"subscribed": False}), 200


def run_notification_thread():
    while True:
        with app.app_context():
            try:
                session = db.session()

                # Query all subscribers and update
                subscribers = session.query(Subscriber).all()
                for subscriber in subscribers:
                    subscriber.update()

                # Close the session
                session.close()

            except Exception as e:
                print(f"Error in notification thread: {str
(e)}")

        time.sleep(3600)


notification_thread = threading.Thread(target=run_notificatio
n_thread)
```

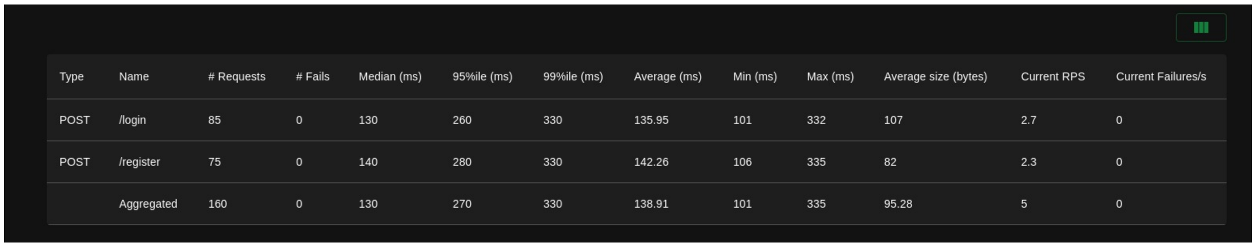
```
notification_thread.daemon = True
notification_thread.start()
```

Firstly, the class `SubscriberModel` describes the subscriber and also provides an `update()` function to get regular weather updates. As far as the `Publisher` is concerned, functions have been given to add, remove & verify subscriptions. Further, a function is present that actually runs in the notifications and updates the weather at the publishing location for every subscriber.

Task 4

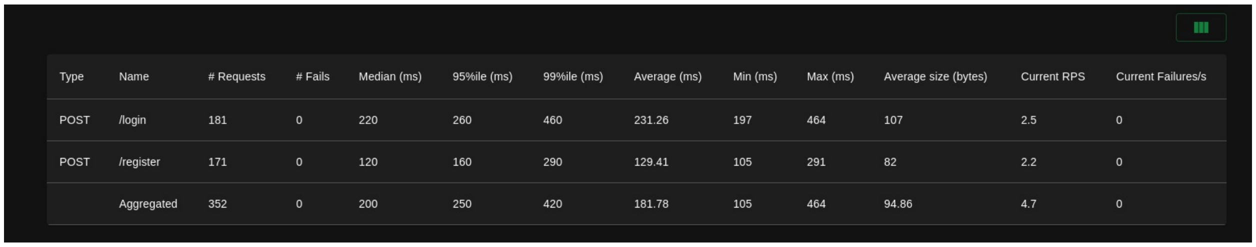
We applied layered architecture for the user login and registration subsytem. The Response time and throughput for each (for 10 users) was obtained using Locust.io

Layered Architecture



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/login	85	0	130	260	330	135.95	101	332	107	2.7	0
POST	/register	75	0	140	280	330	142.26	106	335	82	2.3	0
	Aggregated	160	0	130	270	330	138.91	101	335	95.28	5	0

Microservice Architecture



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/login	181	0	220	260	460	231.26	197	464	107	2.5	0
POST	/register	171	0	120	160	290	129.41	105	291	82	2.2	0
	Aggregated	352	0	200	250	420	181.78	105	464	94.86	4.7	0

The layered architecture approach has more RPS (throughput) of 5 as compared to the Microservice architecture which has RPS of 4.7.

While the layered architecture demonstrates slightly better performance in terms of response time and throughput in this specific scenario, the decision between layered and microservices architectures should consider factors beyond raw performance metrics, such as scalability requirements, development resources, and operational considerations.

Overall, while microservices architecture introduces additional complexity and challenges compared to layered architecture, its advantages in terms of scalability, flexibility, fault isolation, and agility make it a compelling choice for building modern, resilient, and scalable applications.

Link for the code: **https://github.com/vyom2003/TravelApp**