Software Engineering
Assignment 3

Team - 23	
Karan Bhatt	2022202003
Madhusree Bera	2022202007
Vedashree Ranade	2022201073
Piyush Rana	2022202012
Yash Maheshwari	2022201074

Table of Contents

Use Case:

Booking/Ticketing service - For booking tickets to venues/entertainment events. The system shall provide ways to book a ticket based on the number of bookings. This will take into account the total capacity of the venue and the number of tickets that have already been sold.

1. What kind of architectural patterns can be used? Come up with at least 2 patterns.

a. Provide explanation (focus on quality attributes)

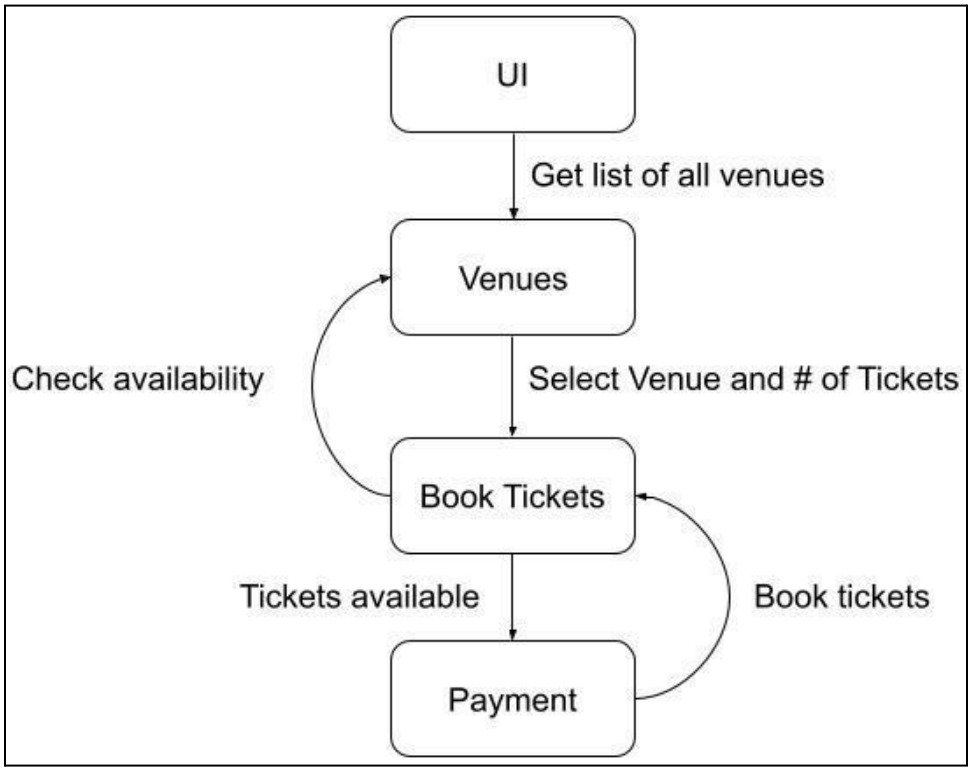
Monolithic:

- **Simpler Development:** Easier to develop and manage initially as all code resides in a single codebase.
- **Faster Initial Deployment:** Less complex infrastructure setup compared to microservices.
- **Easier Data Consistency:** Data is centralized, potentially simplifying data consistency management.

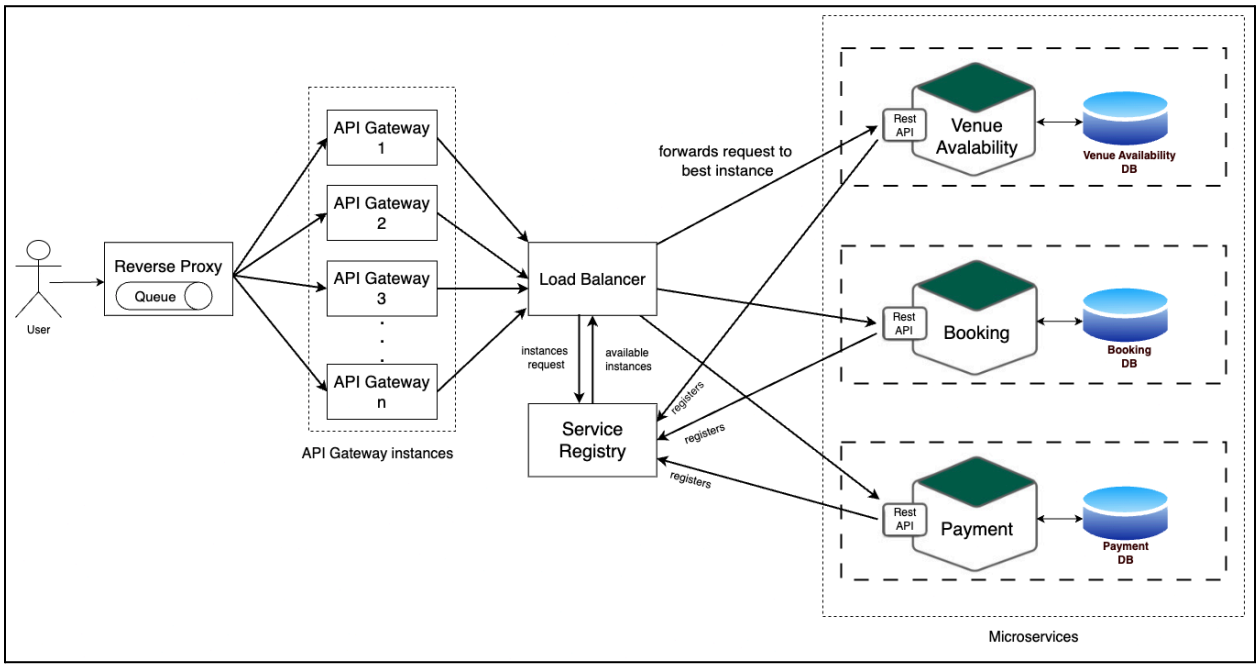
Microservices:

- **Scalability:** Individual services can be scaled independently based on their needs.
- **Modular Development:** Enables independent development and deployment of functionalities by different teams.
- **Improved Maintainability:** Smaller, well-defined services are easier to understand, maintain, and test.
- **Fault Tolerance:** Fault identification is relatively easy in microservices due to their modular architecture, allowing for isolated fault isolation and management.
- **Individual (unit) Testing:** Individual (unit) testing is easier due to the smaller scope and isolated nature of each service, allowing for more focused and effective testing.

b. Create high level diagrams to represent the architecture using the selected patterns.



Monolithic architecture



Microservices architecture

2. Implement both the patterns

a. Proper codebase indicating various services offered along with the relevant patterns.

Monolithic Architecture Code

booking_service.py

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Venue(BaseModel):
    name: str
    capacity: int
    tickets_sold: int

class BookingRequest(BaseModel):
    venue_name: str
    num_tickets: int
    payment_info: float

venues = {
    "Concert Hall": Venue(name="Concert Hall", capacity=5000,
tickets_sold=0),
    "Theater": Venue(name="Theater", capacity=5000, tickets_sold=0),
}

@app.get("/venues")
def get_venues():
    return sorted(list(venues.values()), key=lambda x: x.capacity -
x.tickets_sold, reverse=True)

def payment(payment_info):
    # Simulate payment processing (replace with actual payment gateway
integration)
    # This part would typically involve verifying payment information with
the payment gateway
    # and handling success/failure scenarios.
    print(f"Payment Amount: {payment_info}")
    return True
```

```
@app.post("/book-tickets")
def book_tickets(request: BookingRequest):
    if request.venue_name not in venues:
        return {"message": f"Venue {request.venue_name} is not
registered."}, 404

    venue = venues[request.venue_name]
    if venue.tickets_sold + request.num_tickets <= venue.capacity:
        if payment(request.payment_info):
            venue.tickets_sold += request.num_tickets
            return {"message": f"Successfully booked {request.num_tickets}
tickets for {venue.name}."}
        else:
            return {"message": "Payment failed. Booking not confirmed."},
400
    else:
        return {"message": f"Sorry, there are not enough tickets available
for {venue.name}."}, 400


if __name__ == "__main__":
    import uvicorn

    uvicorn.run("booking_service:app", host="0.0.0.0", port=8000)
```

Booking_client.py

```
import requests

SERVER_URL = "http://localhost:8000"

def get_venues():
    response = requests.get(f"{SERVER_URL}/venues")
    if response.status_code == 200:
        return response.json()
    else:
        print(f"Error getting venues: {response.text}")
        return None

def book_tickets(venue_name, num_tickets):
```

```
    data = {"venue_name": venue_name, "num_tickets": num_tickets,
"payment_info": num_tickets * 500}
    response = requests.post(f"{SERVER_URL}/book-tickets", json=data)
    if response.status_code == 200:
        print((response.json()))
    else:
        print(f"Error booking tickets: {response.text}")

venues = get_venues()
if venues:
    for venue in venues:
        print(f"{venue['name']}: {venue['capacity'] -
venue['tickets_sold']} tickets available")
    venue_name = input("Enter venue name: ")
    num_tickets = int(input("Enter number of tickets: "))
    book_tickets(venue_name, num_tickets)
```

Microservices Architecture Code

- Every component can be distributed and therefore is highly scalable component wise.
- All the communication is done over HTTP in the prototype

In our booking use case, 3 microservices are used which are **Booking Service**, **Payment Service** and **Venue availability service**.

Booking Service:

- Acts as an Orchestrator while using the services of Venue Availability and Payment.
- When a booking request arrives for a venue id, it first checks if the venue is available
- If the venue is available, it confirms the booking and initiates payment request to Payment service
- It waits for the response from payment service and accordingly updates the ‘occupied’ parameter of that Venue by sending a request to Venue Availability Service.
- Finally sends the response of booking to the user

Booking.py

```python
from fastapi import FastAPI, HTTPException
import requests
import os
from dotenv import load_dotenv
import os
from pydantic import BaseModel
import datetime

# Load environment variables from .env file
load_dotenv()


app = FastAPI()
SERVICE_REGISTRY_URL = os.getenv("SERVICE_REGISTRY_URL")
VENUE_AVAILABILITY_SERVICE_NAME = os.getenv("VENUE_AVAILABILITY_SERVICE_NAME")
PAYMENT_SERVICE_NAME = os.getenv("PAYMENT_SERVICE_NAME")

booking_id = 0
class BookingData(BaseModel):
    amount: int
    venue_id: int
    payee_name: str
    payee_account: str

@app.post('/book')
def book(booking_data: BookingData):
    # Fetch venue availability
    #print("Fetching venue availability")
    response =
requests.get(f"{SERVICE_REGISTRY_URL}/find/{VENUE_AVAILABILITY_SERVICE_NAME}")
    if response.status_code != 200:
        raise HTTPException(status_code=500, detail={ "message": "Venue Availability
Service not found"})
    venue_availability_url = response.json()['url']

    response =
requests.get(f"{venue_availability_url}/venue_availability/{booking_data.venue_id}")
    if(response.status_code != 200):
        raise HTTPException(status_code=404, detail={ "message": "Venue not found"})

    venue_availability = response.json()
```

```
    #print(venue_availability)

    if not venue_availability['available']:
        return {"message": "Venue not available"}

    #print("Venue available")

    # Make payment
    response = requests.get(f"{SERVICE_REGISTRY_URL}/find/{PAYMENT_SERVICE_NAME}")
    if response.status_code != 200:
        raise HTTPException(status_code=500, detail={ "message": "Payment System
Service not found"})
    payment_system_url = response.json()['url']

    payment_data = {"amount": booking_data.amount, "payee_name":
booking_data.payee_name, "payee_account": booking_data.payee_account}
    payment_response = requests.post(f"{payment_system_url}/make_payment",
json=payment_data)



    if payment_response.status_code != 200:
        raise HTTPException(status_code=payment_response.status_code,
detail=payment_response.json()['detail'])

    # Update venue occupancy
    venue_data = {"venue_id": booking_data.venue_id}
    response = requests.post(f"{venue_availability_url}/increment_venue_occupancy",
json=venue_data)
    if response.status_code != 200:
        raise HTTPException(status_code=500, detail=response.json()['detail'])
    global booking_id
    booking_id = booking_id + 1

    return {"message": "Booking confirmed", "booking_id": booking_id, "booking_time":
datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S"), "venue_id":
booking_data.venue_id, "amount": booking_data.amount, "payee_name":
booking_data.payee_name}

# Register with Service Registry
def register_with_registry(port):
    service_data = {
        "name": "Booking System",
```

```
        "url": f"http://localhost:{port}"
    }
    requests.post(f"{SERVICE_REGISTRY_URL}/register", json=service_data)

if __name__ == '__main__':
    import uvicorn
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--port", type=int, help="Port number", required=True)
    args = parser.parse_args()
    register_with_registry(args.port)
    #print(f"Booking System running on port {args.port}")
    uvicorn.run(app, host='0.0.0.0', port=args.port)
```

Booking Service API

Venue Availability Service:

- Service to provide information about the venues: venue id, occupancy, availability
- Receives venue id from Booking service and returns the availability status and number of available tickets
- Provides functionality to increment / decrement the available number of tickets at a particular venue by providing venue id

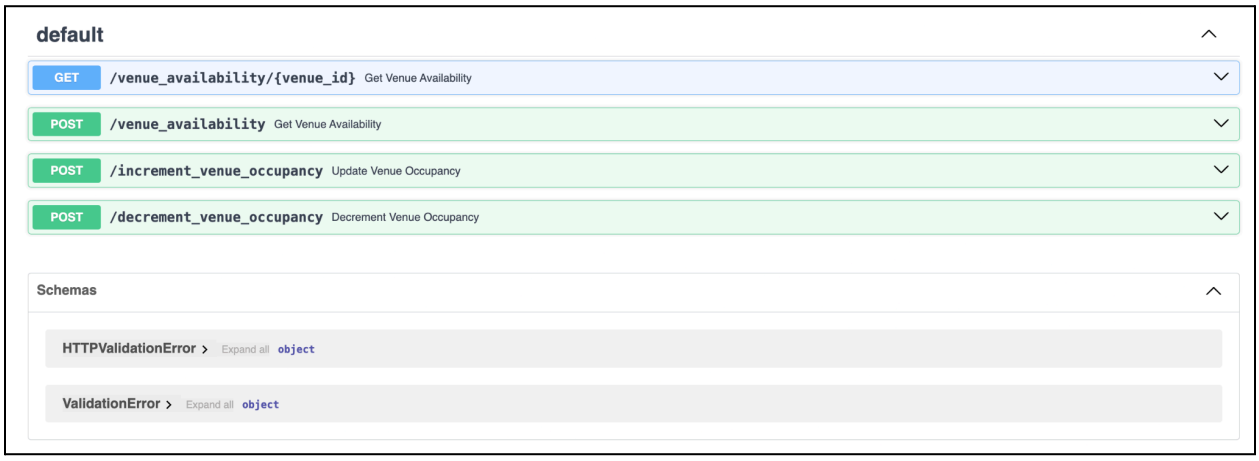
VenueAvailability.py

```python
from fastapi import FastAPI, HTTPException
import requests
from pydantic import BaseModel

from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()




app = FastAPI()
SERVICE_REGISTRY_URL = os.getenv("SERVICE_REGISTRY_URL")

venue_details = [
    {
        "venue_id": 1,
        "capacity": 500,
        "occupied": 0
    },
    {
        "venue_id": 2,
        "capacity": 200,
        "occupied": 0
    },
    {
        "venue_id": 3,
        "capacity": 200,
        "occupied": 0
    }
]

@app.get('/venue_availability/{venue_id}')
def get_venue_availability(venue_id: int):
```

```
    if venue_id < 1 or venue_id > len(venue_details):
        raise HTTPException(status_code=404, detail={"message": "Venue not found"})

    available_seats = venue_details[venue_id - 1]['capacity'] - venue_details[venue_id
- 1]['occupied']
    venue_availability = {
        "venue_id": venue_id,
        "available_seats": available_seats,
        "available": available_seats > 0
    }

    return venue_availability

@app.post('/venue_availability')
def get_venue_availability(venue_data: dict):
    return venue_details
class Venue_Data(BaseModel):
    venue_id: int


@app.post('/increment_venue_occupancy')
def update_venue_occupancy(venue_data: dict):
    venue_id = venue_data['venue_id']
    occupied = venue_details[venue_id - 1]['occupied'] + 1

    if occupied > venue_details[venue_id - 1]['capacity']:
        return {"message": "Venue at full capacity"}
    venue_details[venue_id - 1]['occupied'] = occupied
    return {"message": "Venue occupancy updated"}

@app.post('/decrement_venue_occupancy')
def decrement_venue_occupancy(venue_data: dict):
    venue_id = venue_data['venue_id']
    occupied = venue_details[venue_id - 1]['occupied'] - 1

    if occupied < 0:
        return {"message": "Venue occupancy cannot be negative"}

    venue_details[venue_id - 1]['occupied'] = occupied
    return {"message": "Venue occupancy decremented"}

# Register with Service Registry
```

```python
def register_with_registry(port):
    service_data = {
        "name": "Venue Availability System",
        "url": f"http://localhost:{port}"
    }
    requests.post(f"{SERVICE_REGISTRY_URL}/register", json=service_data)

if __name__ == '__main__':
    import uvicorn
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--port", type=int, help="Port number", required=True)
    args = parser.parse_args()
    register_with_registry(args.port)
    #print(f"Venue Availability System running on port {args.port}")
    #make server restart on code change
    uvicorn.run(app, host='0.0.0.0', port=args.port)
```

Venue Availability API



Payment Service:

- Service to make payment
- Receives payment request from Booking service: amount, payee name, payment account (for simplicity)
- After processing payment, sends response back to the Booking Service

Payment.py

```python
from fastapi import FastAPI, HTTPException
import requests
from dotenv import load_dotenv
from pydantic import BaseModel
import os

# Load environment variables from .env file
load_dotenv()

app = FastAPI()
SERVICE_REGISTRY_URL = os.getenv("SERVICE_REGISTRY_URL")


class PaymentData(BaseModel):
    amount: int
    payee_name: str
    payee_account: str


@app.post('/make_payment')
def make_payment(payment_data: PaymentData):
    # For simplicity, assuming payment is always successful
    if payment_data.amount <= 0:
        raise HTTPException(status_code=401, detail={ "message" : "Invalid amount"})


    success = True
    return {"success": success}

# Register with Service Registry
def register_with_registry(port):
    service_data = {
        "name": "Payment System",
        "url": f"http://localhost:{port}"
    }
    requests.post(f"{SERVICE_REGISTRY_URL}/register", json=service_data)
```

```python
if __name__ == '__main__':
    import uvicorn
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--port", type=int, help="Port number", required=True)
    args = parser.parse_args()
    register_with_registry(args.port)
    #print(f"Payment System running on port {args.port}")
    uvicorn.run(app, host='0.0.0.0', port=args.port)
```

Payment API

Service Registry:

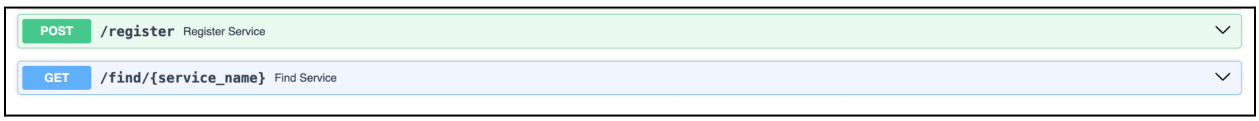
- This is responsible for registering all the available instances of microservices
- Whenever a new instance of any service is started, it sends a request to the Service Registry and registers itself.
- It can also have health monitoring functions

Service Registry provides the most suitable instance of a particular service to other services.

ServiceRegistry.py

```python
from fastapi import FastAPI, HTTPException
import argparse

app = FastAPI()
registered_services = {}

@app.post('/register')
def register_service(service_data: dict):
    #print(f"Registering {service_data['name']} at {service_data['url']}")
    #print(registered_services)
    registered_services[service_data['name']] = service_data['url']
    return {"message": "Service registered successfully"}

@app.get('/find/{service_name}')
def find_service(service_name: str):
    if service_name in registered_services:
        return {"url": registered_services[service_name]}
    else:
        raise HTTPException(status_code=404, detail="Service not found")

if __name__ == '__main__':
    import uvicorn
    parser = argparse.ArgumentParser()
    parser.add_argument("--port", type=int, help="Port number", required=True)
    args = parser.parse_args()
    #print(f"Service Registry running on port {args.port}")
    uvicorn.run(app, host='0.0.0.0', port=args.port)
```

Service Registry API



Load Balancer:

- Load balancer is a private load balancer whose main task is to balance the incoming request and forward it to the most suitable instance of a microservice depending on the load of available instance, geography, etc.

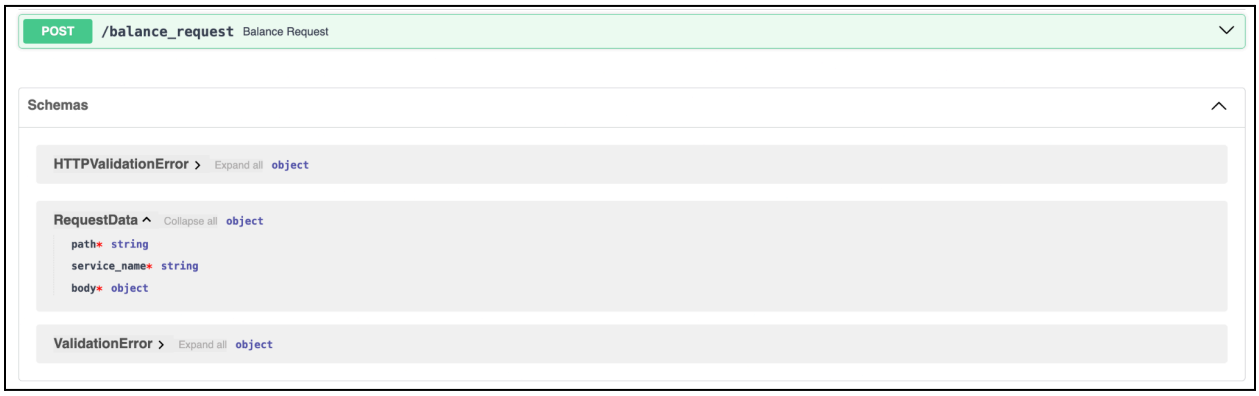
It requests for instances to the Service Registry and forwards API request to that instance

LoadBalancer.py

```python
from fastapi import FastAPI, HTTPException
import requests
from pydantic import BaseModel
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

app = FastAPI()
SERVICE_REGISTRY_URL = os.getenv("SERVICE_REGISTRY_URL")

class RequestData(BaseModel):
    path: str
    service_name: str
    body: dict

@app.post('/balance_request')
def balance_request(request_data: RequestData):
    #print(f"Request directed to load balancer")
    response = requests.get(f"{SERVICE_REGISTRY_URL}/find/{request_data.service_name}")
    if response.status_code != 200:
        raise HTTPException(status_code=404, detail=response.json()['detail'])
    service_url = response.json()['url'] + "/" + request_data.path

    #print(f"Request directed to {request_data.service_name} at {service_url}")
    response = requests.post(f"{service_url}", json=request_data.body)
    return response.json()

if __name__ == '__main__':
    import uvicorn
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--port", type=int, help="Port number", required=True)
    args = parser.parse_args()
```

```
#print(f"Load Balancer running on port {args.port}")
uvicorn.run(app, host='0.0.0.0', port=args.port)
```

Load Balancer API



API Gateway:

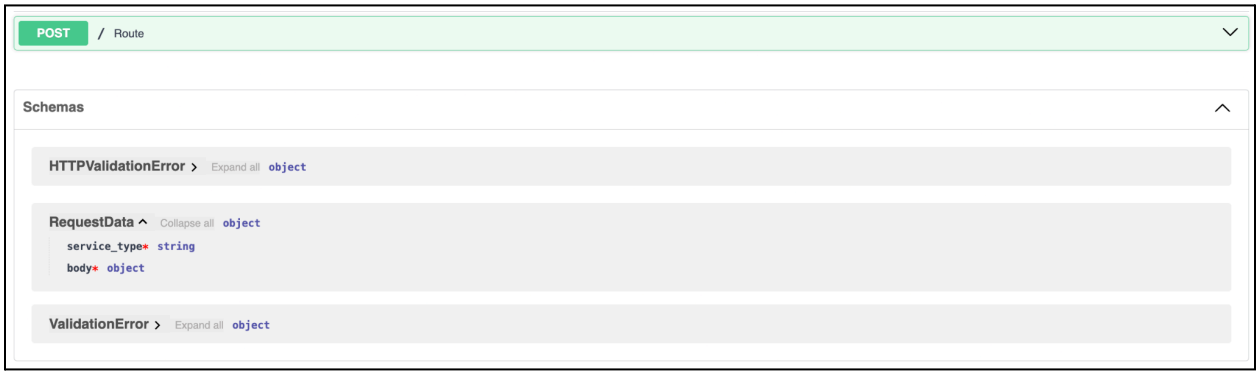
- This receives the API request from the Reverse Proxy and identifies the suitable microservice where the request must be forwarded (request routing)
- It adds the necessary headers, path parameters and body and sends the request to the load balancer
- Multiple instances of API gateways must be provided for scalability
- In our use case, we have tested using 2 API Gateways

APIGateway.py

```python
from fastapi import FastAPI, HTTPException
import requests
from pydantic import BaseModel
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

app = FastAPI()
SERVICE_REGISTRY_URL = os.getenv("SERVICE_REGISTRY_URL")

class RequestData(BaseModel):
    service_type: str
    body: dict

@app.post('/')
def route(request_data: RequestData):
    service_name = ""
    path=""
    if(request_data.service_type == "Payment"):
        #print("Payment API set")
        service_name = os.getenv("PAYMENT_SERVICE_NAME")
        path="make_payment"
    elif(request_data.service_type == "VenueAvailability"):
        #print("Venue Availability API set")
        service_name = os.getenv("VENUE_AVAILABILITY_SERVICE_NAME")
        path="venue_availability"
    elif(request_data.service_type == "Booking"):
        #print("Booking API set")
        service_name = os.getenv("BOOKING_SERVICE_NAME")
        path="book"
```

```
    data = {
        "path": path,
        "service_name": service_name,
        "body": request_data.body
    }
    #print(f"Request directed to load balancer")
    response = requests.post(f"{os.getenv('LOAD_BALANCER_URL')}/balance_request",
json=data)
    return response.json()

if __name__ == '__main__':
    import uvicorn
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--port", type=int, help="Port number", required=True)
    args = parser.parse_args()
    #print(f"API gateway running on port {args.port}")
    uvicorn.run(app, host='0.0.0.0', port=args.port)
```

API Gateway API



Reverse Proxy:

- Entry point for all API to the backend
- It uses round robin algorithm to select the instance of API Gateway to which the request is to be forwarded
- It can also perform security checks on the receiving request and prevent attacks like Denial of Service
- It can also apply rate limiting and other security measures
- The reverse proxy can have a request queue where the incoming requests are stored

ReverseProxy.py

```python
from fastapi import FastAPI, Request, HTTPException
from typing import List
import requests
from pydantic import BaseModel
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()


app = FastAPI()

# Counter to keep track of the current gateway
gateway_counter = 0

class RequestData(BaseModel):
   service_type: str
   body: dict

@app.get("/")
async def read_root():
   return "Reverse Proxy running"


@app.post("/book-tickets")
async def forward_to_gateway(body: RequestData):

   #calculating the gateway index in round robin manner
   global gateway_counter
   gateway_index =( gateway_counter % int(os.getenv("NO_OF_GATEWAYS")) )+1
   gateway_counter += 1
```

```
    # Forward the request to the appropriate gateway
    forward_url = os.getenv(f"API_GATEWAY_URL_{gateway_index}")
    #print(f"Forwarding request to API Gateway {gateway_index}" )

    data = {
        "service_type": body.service_type,
        "body": body.body
    }
    response = requests.post(forward_url, json=data)
    return response.json()


@app.post("/venues")
async def forward_to_gateway(body: RequestData):

    #calculating the gateway index in round robin manner
    global gateway_counter
    gateway_index =( gateway_counter % int(os.getenv("NO_OF_GATEWAYS")) )+1
    gateway_counter += 1


    # Forward the request to the appropriate gateway
    forward_url = os.getenv(f"API_GATEWAY_URL_{gateway_index}")
    #print(f"Forwarding request to API Gateway {gateway_index}" )

    data = {
        "service_type": body.service_type,
        "body": body.body
    }
    response = requests.post(forward_url, json=data)
    return response.json()


if __name__ == "__main__":
    import uvicorn
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--port", type=int, help="Port number", required=True)
    args = parser.parse_args()
    #print(f"Reverse Proxy running on port {args.port}")
```

```
uvicorn.run(app, host='0.0.0.0', port=args.port)
```

Reverse Proxy API

b. Key quality metrics used for comparative analysis.

2.b.1: Performance

Response Time Performance Result (2 minutes):

Monolithic:

1 (max) User | 1 New User per Second

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
GET	/venues	88543	0	0	0.03	0	5	110	1288.2
	Aggregated	88543	0	0	0.03	0	5	110	1288.2

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/book-tickets	131490	0	0	0.05	0	10	77.16	1157.9
	Aggregated	131490	0	0	0.05	0	10	77.16	1157.9

1000 (max) Concurrent Users | 10 New Users per Second

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
GET	/venues	206394	1163	290	307.63	0	746	109.38	1756
	Aggregated	206394	1163	290	307.63	0	746	109.38	1756

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/book-tickets	190090	1236	330	346.72	0	794	77.49	1579.8
	Aggregated	190090	1236	330	346.72	0	794	77.49	1579.8

1000 (max) Concurrent Users | 1000 New Users per Second

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
GET	/venues	209189	8696	580	552.78	15	7183	105.43	1771.3
	Aggregated	209189	8696	580	552.78	15	7183	105.43	1771.3

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/book-tickets	184727	7552	660	617.46	18	8144	74.81	1616.3
	Aggregated	184727	7552	660	617.46	18	8144	74.81	1616.3

Microservices:

1 (max) User | 1 New User per Second

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/venues	13068	0	7	9	22	8.42	6	246	130	110
	Aggregated	13068	0	7	9	22	8.42	6	246	130	110

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/book-tickets	6337	0	15	24	88	17.54	14	184	41.11	56.3
	Aggregated	6337	0	15	24	88	17.54	14	184	41.11	56.3

1000 (max) Concurrent Users | 10 New Users per Second

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/venues	12321	0	5000	9500	14000	5073.82	16	17921	130	104.2
	Aggregated	12321	0	5000	9500	14000	5073.82	16	17921	130	104.2

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/book-tickets	5695	2	9500	18000	25000	9854.19	22	29097	42.03	54.2
	Aggregated	5695	2	9500	18000	25000	9854.19	22	29097	42.03	54.2

1000 (max) Concurrent Users | 1000 New Users per Second

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/venues	6981	49	7500	20000	37000	8392.45	32	46134	129.09	103.3
	Aggregated	6981	49	7500	20000	37000	8392.45	32	46134	129.09	103.3

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/book-tickets	8392	157	14000	26000	42000	12741.48	26	43016	38.6	72.8
	Aggregated	8392	157	14000	26000	42000	12741.48	26	43016	38.6	72.8

Comparison Table

/book-tickets

Test Details	**Avg time in Monolithic Architecture (ms)**	**Avg time in Microservices Architecture (ms)**
1 (max) User | 1 New User per Second	0.05	17.54
1000 (max) Concurrent Users | 10 New Users per Second	346.72	9854.19
1000 (max) Concurrent Users | 1000 New Users per Second	552.78	12741.48

/venues

Test Details	**Avg time in Monolithic Architecture (ms)**	**Avg time in Microservices Architecture (ms)**
1 (max) User | 1 New User per Second	0.03	8.42
1000 (max) Concurrent Users | 10 New Users per Second	307.63	5073.82
1000 (max) Concurrent Users | 1000 New Users per Second	617.46	12741.48

2.b.2 Scalability

Monolithic

- Scaling requires **replicating the entire application stack**, limiting flexibility in resource allocation.
- Vertical scaling (upgrading hardware) is the primary method, often leading to resource inefficiency.
- Horizontal scaling is **challenging** due to tight coupling between components.

Microservices

- Horizontal scaling is easier as individual services can be **scaled independently** based on demand.
- Granular resource allocation allows for **efficient utilization of resources**.

- **Autoscaling** is more feasible due to the independent nature of services.

2.b.3 Modifiability

Monolithic

- Modifications require changes to a single codebase, leading to **longer build and deployment cycles**.
- Dependencies between modules can result in unintended consequences (**ripple effect**) and increase resistance to change over time.
- Adopting new technologies or frameworks for specific tasks is **more difficult** due to the monolith's cohesive nature.

Microservices

- Modifications are localized to individual services, **enabling faster iteration and deployment cycles**.
- **Loose coupling** between services reduces the risk of unintended consequences when making changes.
- Adoption of new technologies or frameworks is **easier** as services can use different technologies as needed.

2.b.4 Testability

Monolithic

- Testing integration and scalability can be **complex** due to the monolith's tightly coupled nature.
- Longer build and deployment cycles slow down the iteration process, affecting test frequency.
- Identifying and **isolating issues may be challenging**, especially in large and complex codebases.

Microservices

- Modifications are localized to individual services, enabling **faster iteration and Testing individual services** in isolation is simpler, enabling faster identification and isolation of issues.
- Frequent iteration and deployment cycles support **continuous testing practices**, improving overall system quality.
- Testing scalability and performance is **more straightforward** as services can be independently tested and scaled.

3. Which pattern according to you performs better? Why?

Table showing Comparative Analysis

Quality Metric	**Better Architecture**
Performance	Monolithic Architecture
Scalability	Microservices Architecture
Modifiability	Microservices Architecture
Testability	Microservices Architecture

As can be seen, Monolithic is performing better as it is taking less time to process the request for the given booking scenario. However, there will be multiple services including IoT system, Parking lot management, Recommendation system, etc. in the original application. Implementing a Monolithic architecture in such a scenario will make scalability, modifiability and testability very challenging. Hence, Microservices architecture will be suitable. Also it must be noted that the performance of our prototype is dependent on the available resources in our system. In reality, Microservices can also provide sufficiently good performance. Thus, in conclusion, taking into account several quality metrics, Microservices will be a better architecture choice.