# Software Engineering
# Group Project: 3

## Casino Management System (CMS)

***TEAM: 17***

Ashutosh Srivastava
Manuj Garg
Rayaan Khan
Yash Shivhare
Yatharth Gupta

# CONTENT

# SUBSYSTEM OVERVIEW

## User Management

- The system manages user registration, authentication, and authorization. Passwords are securely stored using bcrypt. Admins can add new managers, who can then login and perform tasks like creating casinos.

---

## Casino Management

- Handles casino information, financial data, and game table performance.
- Enables casino creation using a builder pattern, with input for the number of game tables, bars, and staff if you're a manager.
- Lists all casinos for users, allowing them to view specific casino details, including game tables and bars. Users can interact with game tables, leading to potential profit or loss, which is stored in the database.

---

## Game Management

- Configures and tracks game tables and gameplay data.

## Payment Processing

- Handles chip purchases, and redemptions, and integrates with external payment gateways.
- The entire subsystem is built on Strategy and Adapter pattern for different modes of payments.

## Notification

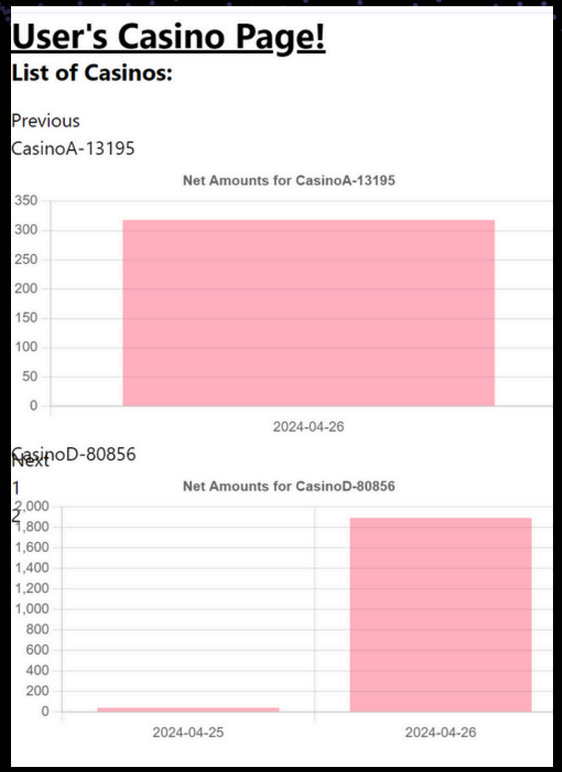
- User can subscribe to any casino, and casino can notify these users for any new updates.

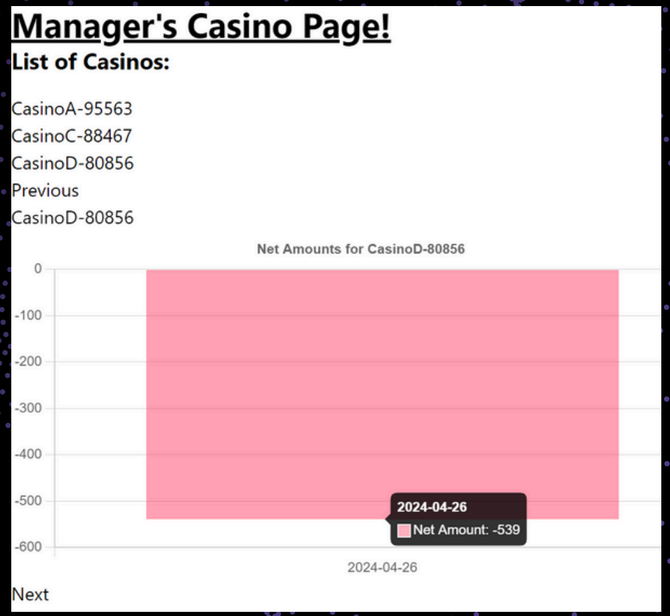
# SUBSYSTEM OVERVIEW

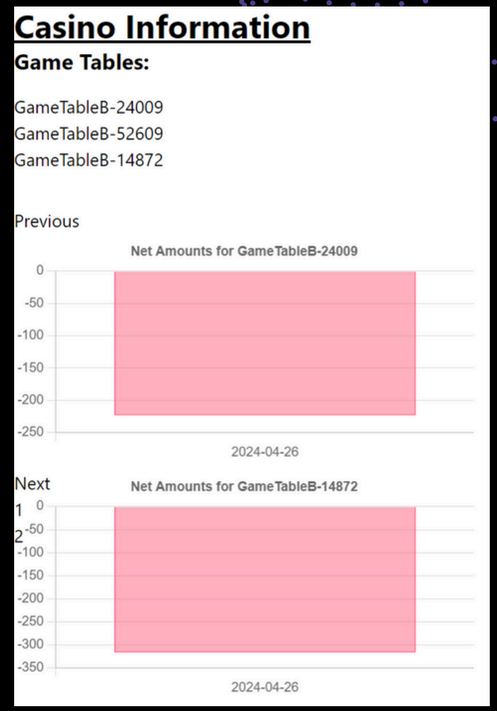
## Analytics

- **User**: Analytics based on a particular day and in terms of usability, easily keep track of his overall expenses in different casinos.

- **Manager:** Can analyze overall profit/loss across all casinos under their supervision. They can identify the most profitble casinos and track daily trends. Additionally, they can detect abnormal patterns and assess which game tables are generating the most profits.



# FUNCTIONAL REQUIREMENTS

## User Management

- Register and manage customer accounts
- Manage employee accounts and roles
- Track user activity and gameplay data

## Casino Management

- Sub-brands and individual casino information
- Financial data (income, expenses, profits)
- Game table performance and profitability

## Game Table Management

- Configure and manage different types of game tables
- Track game table usage and profitability

## Chips Management

- Facilitate chip purchase and redemption

## Payment Processing

- Support cash and various electronic payment methods (UPI, debit card)

## Notifications

- Send notifications to subscribed customers of a casino.

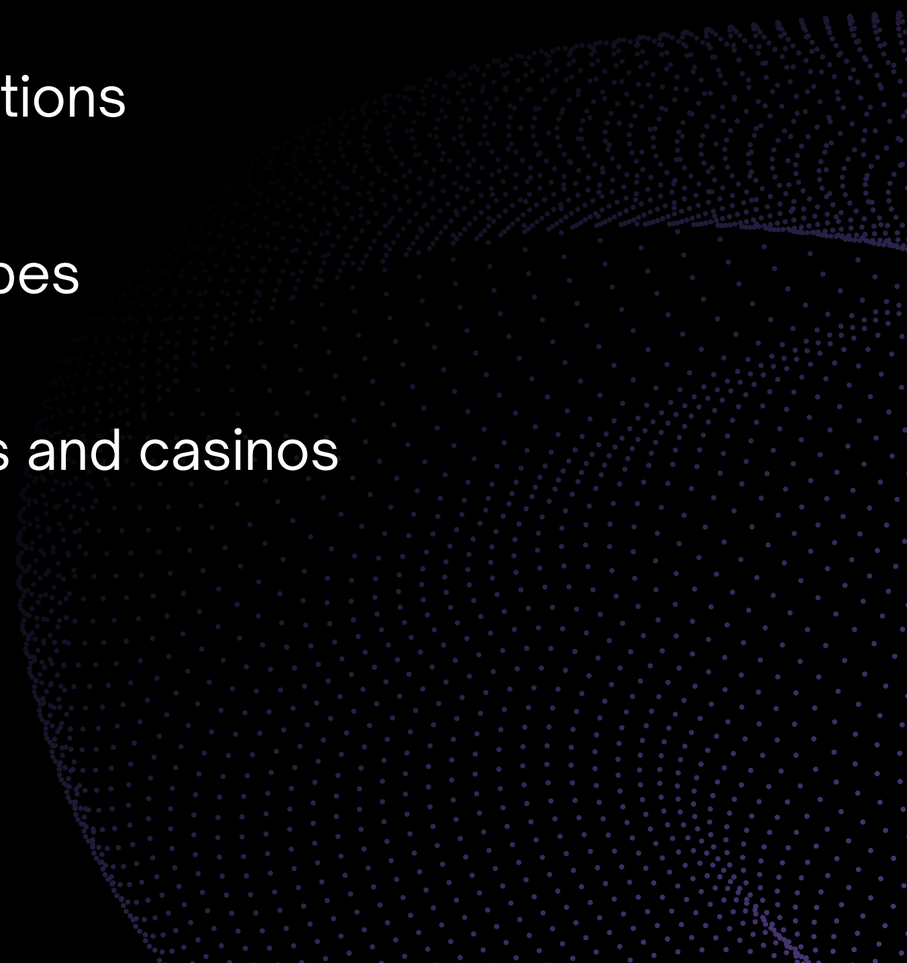
# NON-FUNCTIONAL REQUIREMENTS

**Security**: Protect user data from unauthorized access

**Performance:** Ensure fast response times and efficient operations

**Extendibility:** Easily add new casino sub-brands and game types

**Usability:** Provide a user-friendly interface for both customers and casinos



# 2 ARCHITECTURAL FRAMEWORK

## 2.1 Stakeholder Identification

| Stakeholder | Concerns | Viewpoint/View |
| --- | --- | --- |
| Customers | Security, usability, performance | User Interface, System Performance |
| Casino Operators | Profitability, efficiency, data insights | Management Dashboards, Financial Reports |
| Developers | Code quality, maintainability, extensibility | System Design, Code Structure |

# 2 ARCHITECTURAL FRAMEWORK

## 2.2 Major Design Decisions (ADRs)

| ADR | Decision | Rationale |
|---|---|---|
| 1 | Use Python, and React | Familiar technologies, suitability for project requirements |
| 2 | Implement Layered Architecture pattern | Not a huge system, modularity, maintainability so Layered was preferred over MicroService |
| 3 | Utilize a single central database | Simplifies data management, ensures data consistency , improved data access , enhances security and compliance |
| 4 | Employ RESTful API for communication | Standardized interface, flexibility for future integrations,loose coupling between client and server,more secure ( used JWT(in our case)) |
| 5 | Choice of database | SQL over MongoDB as most of our data is relational so SQL is a better fit for relational data.The analytics can be done easily when we use SQL over MongoDb |

# 3 Architectural Tactics and Patterns

## 3.1 Architectural Tactics

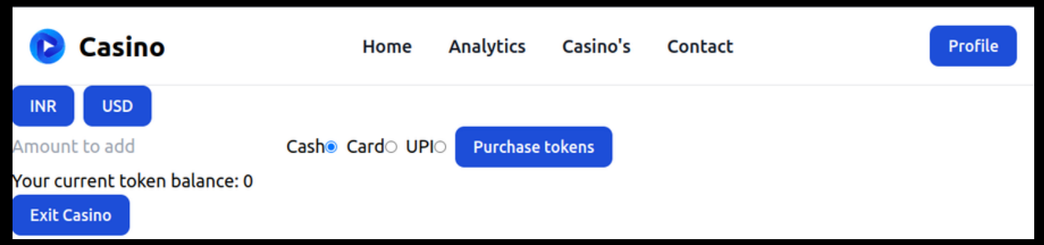
- **Security Tactics**
  - User Authentication: Verify user credentials using bcrypt.
  - User Authorization: Utilize tokens stored in local storage to fetch user ID and roles for backend calls.
  - Data Confidentiality: Encrypt passwords with bcrypt for enhanced security.

- **Availability Tactics**
  - Exception Handling: System gracefully handles errors from backend API calls, ensuring continuous functionality and responsiveness.

- **Modifiability Tactics**
  - Semantic Coherence: Maintain clear and consistent code structure for easy modifications and adaptability.

# 3.2 Implementation Patterns

# Adapter Pattern



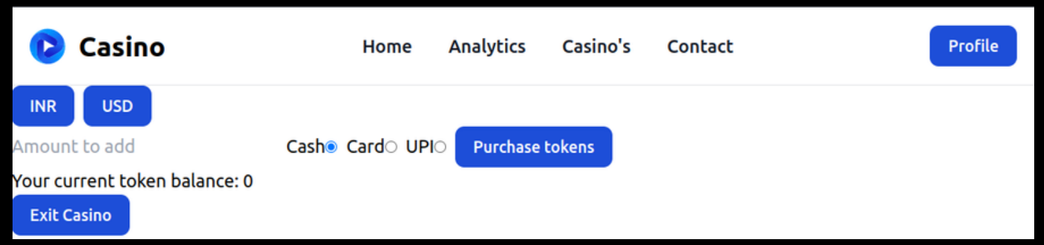

```python
class CurrencyConverter:
    def convert_to_usd(self, amount):
        return int(amount)*83

class PaymentStrategyAdapter(PaymentStrategy):
    def __init__(self, adaptee, currency_converter):
        self.adaptee = adaptee
        self.currency_converter = currency_converter

    def pay(self, amount):
        amount_usd = self.currency_converter.convert_to_usd(amount)
        print(amount_usd)
        self.adaptee.pay(amount_usd)
        return amount_usd

    def authorize(self):
        self.adaptee.authorize()
```

We have a class Currency Converter defined which is used to convert the USD$ to Indian Rupees. The Class PaymentStrategyAdapter is the adapter class.

```python
def process_payment(user_id, amount, payment_method,currency="INR"):
    # Select payment strategy based on the chosen method
    if payment_method == 'cash':
        payment_context = CashPayment()
        if currency == "USD":
            currency_converter = CurrencyConverter()
            payment_context = PaymentStrategyAdapter(payment_context, currency_converter)
    elif payment_method == 'card':
        payment_context = CardPayment()
    elif payment_method == 'upi':
        payment_context = UpiPayment()
    else:
        return {"status": "error", "message": "Invalid payment method"}

    # Make payment using the selected strategy
    payment_context.authorize()
    final_amt=payment_context.pay(amount)
    return final_amt
```

In the process_payment function if the payment method is cash we use the CashPayment as paymentcontext. If the currency used is USD$ , we wrap the adapter PaymentStrategyAdpater around the payment context(CashPayment) . When we call pay method it directs to the method pay defined in the adapter which first converts the USD dollars to Indian Rupees and then that pay function is executed which is done in INR case.

# Strategy Pattern




```python
from app.models.builder.PaymentStrategy import PaymentStrategy

class CashPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paying {amount} using Cash.")
        return amount
    def authorize(self):
        print("Cash payment authorized")

class CardPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paying {amount} using Credit/Debit Card.")
        return amount
    def authorize(self):
        print("Authorizing payment...")

class UpiPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paying {amount} using UPI.")
        return amount
    def authorize(self):
        print("Authorizing payment...")
```

Concrete Strategies

```python
from abc import ABC, abstractmethod

class PaymentStrategy(ABC):
    @abstractmethod
    def pay(self, amount):
        pass
    def authorize(self):
        pass
```

Payment Strategy

```python
import PaymentStrategy
class PaymentContext:
    def __init__(self, strategy: PaymentStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: PaymentStrategy):
        self._strategy = strategy

    def execute_payment(self, amount):
        self._strategy.authorize()
        self._strategy.pay(amount)
```

Payment Context

```python
@app.route('/wallet/addBalance', methods=['POST'])
def add_balance():
    user_id = request.json['user_id']
    amount = request.json['amount']
    strategy = request.json['strategy']
    currency = request.json['currency']
    token_wallet_dao = TokenWalletDao()
    earlier_balance = token_wallet_dao.get_wallet_balance(user_id)
    final_amt=process_payment(user_id, amount, strategy,currency)
    total = earlier_balance + int(final_amt)

    token_wallet_dao.update_wallet_balance(user_id, total)
    return jsonify({"status": "success"})
```

```python
from app.dao.TokenWalletDao import TokenWalletDao
from app.models.builder.ConcreteStrategy import CashPayment,CardPayment,UpiPayment
from app.models.builder.USDAdapter import CurrencyConverter,PaymentStrategyAdapter
# Function to handle payment
def process_payment(user_id, amount, payment_method,currency="INR"):
    # Select payment strategy based on the chosen method
    if payment_method == 'cash':
        payment_context = CashPayment()
        if currency == "USD":
            currency_converter = CurrencyConverter()
            payment_context = PaymentStrategyAdapter(payment_context, currency_converter)
    elif payment_method == 'card':
        payment_context = CardPayment()
    elif payment_method == 'upi':
        payment_context = UpiPayment()
    else:
        return {"status": "error", "message": "Invalid payment method"}
```

The payment strategy system promotes code reusability and maintainability by encapsulating payment method-specific logic within separate strategy classes.

# Observer Pattern

We have implemented Observer Pattern to send notifications to users

## The flow of functionality:

- Users subscribe/unsubscribe to a particular casino: The system checks if the subscription already exists and performs the opposite action accordingly.

```python
@app.route('/subscribe', methods=['POST'])
@jwt_required()
def subscribe():
    print("here")
    userId = get_jwt_identity()
    casinoId = request.json['casinoId']

    casino = casino_dao.get_casino(casinoId)
    if( casino.check_subscription(userId)):
        casino.detach(userId)
        return jsonify({'status': 'unsubscribed'})
    else:
        casino.attach(userId)
        return jsonify({'status': 'subscribed'})
```

```python
@app.route("/check_subscription", methods=['POST'])
@jwt_required()
def check_subscription():
    userId = get_jwt_identity()
    casinoId = request.json['casinoId']
    print(userId)
    print(casinoId)
    casino = casino_dao.get_casino(casinoId)
    if( casino.check_subscription(userId)):
        return jsonify({'status': 'subscribed'})
    else:
        return jsonify({'status': 'unsubscribed'})
```

# Observer Pattern

- Casino sends notification: The manager can enter the message and click Notify to send.

| gameTableC | gameTableD |
|---|---|
| GameTableC-39060 | GameTableD-3520 |
| GameTableC-27478 | GameTableD-47351 |
| GameTableC-69865 | |
| **Bars** | |
| Bar-81255 | |
| **TokenCounter** | |
| tokencounter_e63e4bbf-3580-4b1e-9025-e129c459325e | |

Enter your notification here | Notify

```
const handleNotifyClick = async () => {
  try {
    const response = await fetch("/notify", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        casinoId: casinoId,
        text: text,
        managerId: managerId,
      }),
    });
    if (!response.ok) {
      throw new Error("Failed to submit form data");
    }
    const data = await response.json();
    console.log(data);
    // Handle successful response if needed
    console.log("Notified successfully");
  } catch (error) {
    console.error("Error notifying:", error);
  }
};
```

call from frontend

```
@app.route('/notify', methods=['POST'])
def notify():
    print("Notifying subsribers")
    casinoId = request.json['casinoId']
    managerId = request.json['managerId']
    text = request.json['text']

    casino = casino_dao.get_casino(casinoId)
    casino.send_notification(text)

    return jsonify({'status':'Success'})
```

backend receives the call in CasinoResource.py

```
def attach(self, userId):
    userid = SubscriptionDao.create_user_subscription(userId, self.__casinoid)
    if(userid):
        print(f"User {userId} subscribed to casino {self.__casinoid}")

def detach(self, userId):
    userid = SubscriptionDao.remove_user_subscription(userId, self.__casinoid)
    if(userid):
        print(f"User {userId} unsubscribed from casino {self.__casinoid}")
```

add/remove subscribers from a casino

```
from app import get_db_connection
# make a static SubscriptionDao class
class SubscriptionDao:
    @staticmethod
    def get_subscribers_by_casino_id(casinoid):
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT userid FROM user_subscription WHERE casinoid = ?", (casinoid,))
        subscriber_ids = [row[0] for row in cursor.fetchall()]
        conn.close()
        return subscriber_ids

    @staticmethod
    def create_user_subscription( userid, casinoid):
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("INSERT INTO user_subscription (userid, casinoid) VALUES (?, ?)", (userid, casinoid))
        conn.commit()
        conn.close()
        return userid

    @staticmethod
    def check_user_subscription(userid, casinoid):
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM user_subscription WHERE userid=? AND casinoid=?", (userid, casinoid))
        user_subscription = cursor.fetchone()
        conn.close()
        if user_subscription:
            return True
        return False

    @staticmethod
    def remove_user_subscription(userid, casinoid):
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("DELETE FROM user_subscription WHERE userid=? AND casinoid=?", (userid, casinoid))
        conn.commit()
        conn.close()
        return userid
```

SubscriptionDao

```
def notify(self, message):
    subscriptionIds = SubscriptionDao.get_subscribers_by_casino_id(self.__casinoid)
    for subscriptionId in subscriptionIds:
        print(f"Sending notification to user {subscriptionId}")
        user = user_dao.get_user_by_id(subscriptionId)
        if(type(user) is User):
            user.update(message, self.__casinoid)

def send_notification(self, message):
    print(f"Casino {self.__casinoid} is sending notification: {message}")
    self.notify(message)
```

send notifications function in Casino class

```
def update(self, message, casino_id):
    # save notification in database
    user_dao.add_notification_to_db(self.__id, message, casino_id)


    def add_notification_to_db(self, userId, message, casino_id):
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("INSERT INTO notifications (userid, message,
casinoid) VALUES (?, ?, ?)", (userId, message, casino_id))
        conn.commit()
        conn.close()
        return userId
```

adds the notification to db.

# Observer Pattern

- User gets the notification: The user can go to the notification tab to see all the notifications from the casinos he has subscribed to.

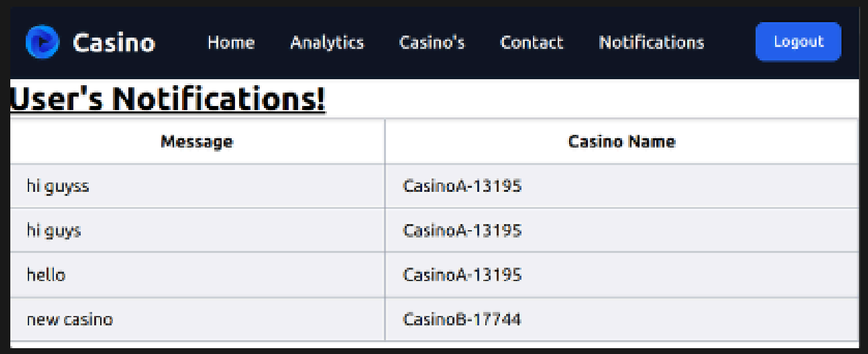
```
useEffect(() => {
  async function fetchNotifications() {
    try {
      const response = await fetch("/get_user_notifications", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({ userId: userId }),
      });
      if (!response.ok) {
        throw new Error("Failed to fetch notifications");
      }
      const data = await response.json();
      console.log(data.notifications);
      setNotifications(data.notifications);
    } catch (error) {
      console.error("Error fetching notifications:", error);
    }
  }

  fetchNotifications();
}, [userId]);
```

Fetch all user notifications

```
@app.route('/get_user_notifications', methods=['POST'])
def get_user_notifications():
    userId = request.json['userId']
    notifications = user_dao.get_user_notifications(userId)
    print(notifications)
    # to return the notifications first we need to convert
    notifications_list = []
    for notification in notifications:
        notification_dict = {
            "message": notification[0],
            "casinoname": notification[1]
        }
        notifications_list.append(not
    return jsonify({"notifications": notifications_list})
```

backend receives the request in UserResource.py

```
    def (variable) cursor: Cursor , userId):
        cursor = conn.cursor()
        # join the notifications table with the
        # casino_token_mg table to get the casinoname
        cursor.execute("SELECT n.message, c.casinoname
FROM notifications n JOIN casino_token_mg c ON n.casinoid
= c.casinoid WHERE n.userid=?", (userId,))
        notifications = cursor.fetchall()
        conn.close()
        return notifications
```




User interface for viewing notifications

# Builder Pattern

We have implemented Builder Pattern for creation of casino

- In our project we have created 4 different types of casinos as described below:
  - Casino A contains: Game table A, C
  - Casino B contains: Game table A, B
  - Casino C contains: Game table C, D
  - Casino D contains: Game table B, D

- We have created 4 types of game tables as described below:
  - Game table A has: Probability: 0.3, Type: Dice
  - Game table B has: Probability: 0.7, Type: Card
  - Game table C has: Probability: 0.5, Type: Card
  - Game table D has: Probability: 0.5, Type: Dice

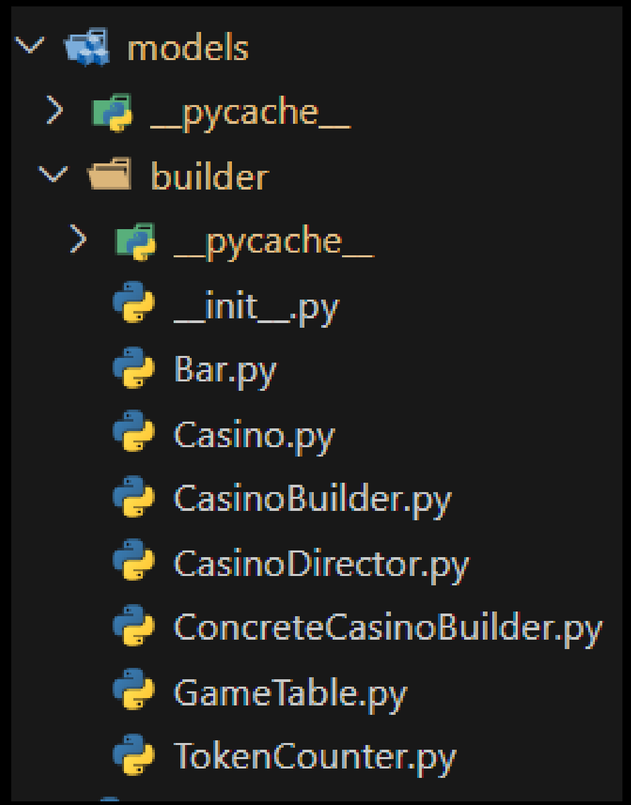
More different kinds of Game tables and Casinos can be created accordingly.

# Builder Pattern - Steps

This is the flow of the creation of Casino using builder pattern

1. A frontend path is defined where we retrieve the user input.
2. We get the stafflist which doesn’t have any gametable or bar assigned to them.
3. We create a CasinoBuilder and pass it to the CasinoDirector.
4. Then we call the corresponding constructCasino{type}() function which is implemented in the director.
5. After this we call the .getResult() function to get the final built Casino class object.

# File structure & CasinoBuilder Class




```python
class CasinoBuilder():

    @abstractmethod
    def constructGameTableA(number):
        pass

    @abstractmethod
    def constructGameTableB(number):
        pass

    @abstractmethod
    def constructGameTableC(number):
        pass

    @abstractmethod
    def constructGameTableD(number):
        pass

    @abstractmethod
    def constructBar():
        pass

    @abstractmethod
    def constructTokenCounter():
        pass
```

# Calling Casino Builder

```python
@app.route('/casino/add',methods=['POST'])
def add_casino():
    print(request.json)
    managerId = request.json['userId']
    casinoType = request.json['casinoType']
    tableA = request.json['gameTableA']
    tableB = request.json['gameTableB']
    tableC = request.json['gameTableC']
    tableD = request.json['gameTableD']
    num_bar = request.json['bar']
    print("casinoType: ", casinoType)
    stafflist = user_dao.get_staff_list(tableA+tableB+tableC+tableD+num_bar)
    staffid_list = [row['staffid'] for row in stafflist]
    print("length of stafflist: ", len(staffid_list))
    builder: CasinoBuilder = ConcreteCasinoBuilder()
    director = CasinoDirector(builder, tableA, tableB, tableC, tableD, num_bar, staffid_list, casinoType)
    if(casinoType=='A'):
        director.constructCasinoA()
    elif(casinoType=='B'):
        director.constructCasinoB()
    elif(casinoType=='C'):
        director.constructCasinoC()
    elif(casinoType=='D'):
        director.constructCasinoD()
    else:
        print("Invalid casino")
        return jsonify({'id': -1})
    casino = builder.getResult(managerId, casinoType)
    if(type(casino) is Casino):
        return jsonify({'id': casino.get_casinoid(), 'status': 'Success'})
    else:
        return jsonify({'status': 'Failed'})
```

# ConcreteCasinoBuilder & CasinoDirector

```python
class ConcreteCasinoBuilder(CasinoBuilder):

    def __init__(self):
        self.TableA = []
        self.TableB = []
        self.TableC = []
        self.TableD = []
        self.Bar = []
        self.StaffId = []
        self.TokenCounterId = None
        self.TokenCounterDao = TokenCounterDao()
        self.StaffDao = StaffDao()
        self.GameTableDao = GameTableDao()
        self.BarDao = BarDao()
        self.CasinoDao = CasinoDao()

    def constructGameTableA(self,number,staffid):

        for i in range(number):
            if(i >= len(staffid)):
                tableId = self.GameTableDao.create_gametable("-1", 0.3, "dice", "A")
            else:
                tableId = self.GameTableDao.create_gametable(staffid[i], 0.3, "dice", "A")
                self.StaffDao.update_assignedId(staffid[i], tableId)
                self.StaffId.append(staffid[i])
            self.TableA.append(tableId)

    def constructGameTableB(self,number,staffid):
        for i in range(number):
            if(i >= len(staffid)):
                tableId = self.GameTableDao.create_gametable("-1", 0.7, "card", "B")
            else:
                tableId = self.GameTableDao.create_gametable(staffid[i], 0.7, "card", "B")
                self.StaffDao.update_assignedId(staffid[i], tableId)
                self.StaffId.append(staffid[i])
            self.TableB.append(tableId)
```

```python
class CasinoDirector:

    def __init__(self, builder, tableA, tableB, tableC, tableD, num_bar, stafflist, casinoType):
        self.casinobuilder = builder
        self.tableA = tableA
        self.tableB = tableB
        self.tableC = tableC
        self.tableD = tableD
        self.num_bar = num_bar
        self.staffid = stafflist
        self.casinoType = casinoType

    def constructCasinoA(self):
        staffid1 = []
        staffid2 = []
        staffid3 = []
        for i in range(len(self.staffid)):
            if(i < self.tableA):
                staffid1.append(self.staffid[i])
            elif(i >= self.tableA and i < self.tableA+self.tableC):
                staffid2.append(self.staffid[i])
            else:
                staffid3.append(self.staffid[i])
        self.casinobuilder.constructGameTableA(self.tableA,staffid1)
        self.casinobuilder.constructGameTableC(self.tableC,staffid2)
        self.casinobuilder.constructTokenCounter()
        self.casinobuilder.constructBar(self.num_bar,staffid3)
```

```python
def getResult(self, managerId, casinoType):
    TablesList = self.TableA + self.TableB + self.TableC + self.TableD
    casino_id = "casino" + casinoType + "_" + str(uuid.uuid4())
    casino = self.CasinoDao.add_casinoTokenMg(casino_id, self.TokenCounterId, managerId, casinoType)
    for gameTableid in TablesList:
        self.CasinoDao.add_casinogametable(casino_id, gameTableid)
    for barId in self.Bar:
        self.CasinoDao.add_casinobar(casino_id, barId)

    return casino
```

# SQLite Queries

```python
def add_casinoTokenMg(self, casinoId, tokenCounterId, managerId, casinoType):
    conn = get_db_connection()
    cursor = conn.cursor()
    random_number = random.randint(0, 100000)
    casinoName = "Casino" + casinoType + "-" + str(random_number)
    tokenCounterName = "TokenCounter" + "-" + str(random_number)
    casino = Casino(casinoId, casinoName, tokenCounterName, tokenCounterId, managerId)
    cursor.execute("INSERT INTO casino_token_mg (casinoid, casinoname, tokencountername, tokencounterid, managerid) VALUES (?, ?, ?, ?, ?)", (casinoId, casinoName, to
    conn.commit()
    conn.close()
    return casino

def add_casinogametable(self, casinoId, gameTableId):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO casino_gametable (casinoid, gametableid) VALUES (?, ?)", (casinoId, gameTableId))
    conn.commit()
    conn.close()
    return

def add_casinobar(self, casinoId, barId):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO casino_bar (casinoid, barid) VALUES (?, ?)", (casinoId, barId))
    conn.commit()
    conn.close()
    return
```

# SQLite Queries Contd.

```python
def get_staff_list(self, number):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT staffid FROM staff WHERE currentassignedid = '-1' ORDER BY currentassignedid LIMIT ?", (number,))
    staff_list = cursor.fetchall()
    conn.close()
    return staff_list
```

```python
def update_assignedId(self, staff_id, assigned_id):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("UPDATE staff SET currentassignedid=? WHERE staffid=?", (assigned_id, staff_id))
    conn.commit()
    conn.close()
    return staff_id
```

```python
def create_gametable(self, staffid, prob, type, tableType):
    gametableid = "gametable" + tableType + "_" + str(uuid.uuid4())
    random_number = random.randint(0, 100000)
    name = "GameTable" + tableType + "-" + str(random_number)
    gameTable = GameTable(gametableid, name, staffid, prob, type)
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("INSERT INTO gametable (gametableid, name, staffid, prob, type) VALUES (?, ?, ?, ?, ?)", (gameTable.get_id(), gameTable.get_name(), gameTable.get_s
    conn.commit()
    conn.close()
    return gameTable.get_id()
```

# Prototype Implementation and Analysis

## Layered Architecture

Divide the system into presentation, service, data access, and persistence layers for a clear separation of concerns. The directory structure for the layered architectural pattern is given below:

The layers present in our System are:

1. Presentation: Frontend
2. Service: Resource files
3. Data Access: DAO files
4. Persistence layers: Database (.db file - using SQLite3)

# Microservice Architecture

Architectural approach to build software applications as a collection of small, independent services, each focused on a specific business function and communicating via APIs.

We have a middleman which is running on port 5000. Basically all the redirects that the frontend does to the backend is directed to the functions defined in the middleman.

```
if __name__ == '__main__':
    app.run(port=5000)
```

# Microservice Architecture

This is a snippet of function defined to handle get request at the path ‘/wallet/balance’ which basically redirects it into the microservice which handles payment functionality and return the response.

```python
@app.route('/wallet/balance',methods=['GET'])
def handle_payment():
    print("hello")
    # Forward the payment request to the payment backend
    payment_response = requests.get('http://localhost:8080/wallet/balance', json=request.json,headers={'Content-Type': 'application/json'})
    print(payment_response.json())
    return payment_response.json(), payment_response.status_code
```

Similarly below is the function to handle the post request at the path ‘/casino/add’ which is redirected to the microservice which handles the non payment functionality.

```python
@app.route('/casino/add', methods=['POST'])
def handle_casino_add():
    print("hello")
    # Forward the payment request to the payment backend
    payment_response = requests.post('http://localhost:5001/casino/add', json=request.json,headers={'Content-Type': 'application/json'})
    print(payment_response.json())
    return payment_response.json(), payment_response.status_code
```

# Microservice Architecture

The other subsystem’s microservice is running on the port 5001

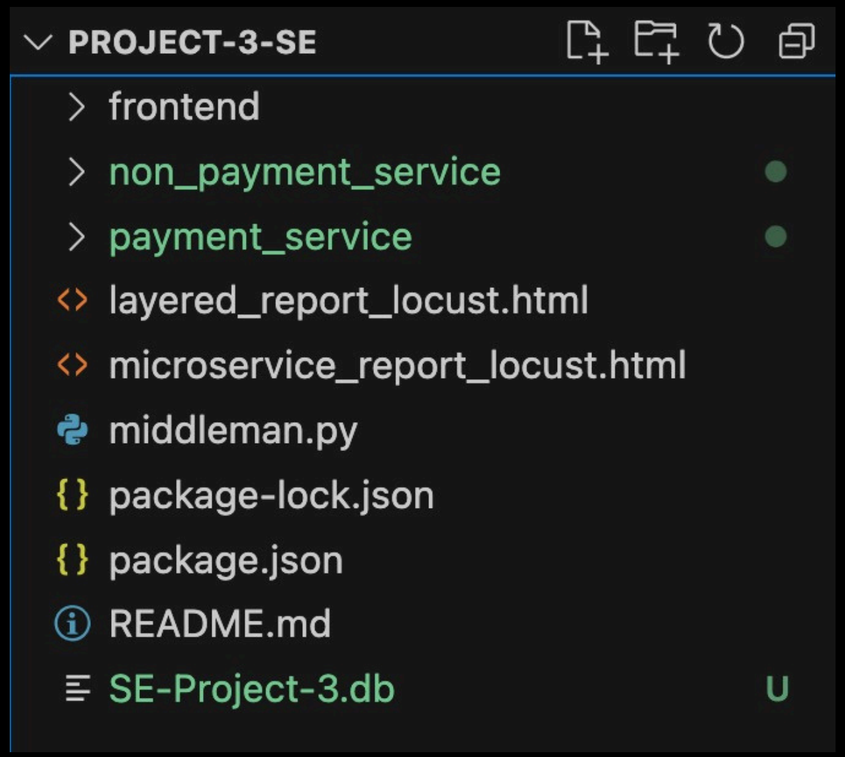
```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0',port=5001)
```

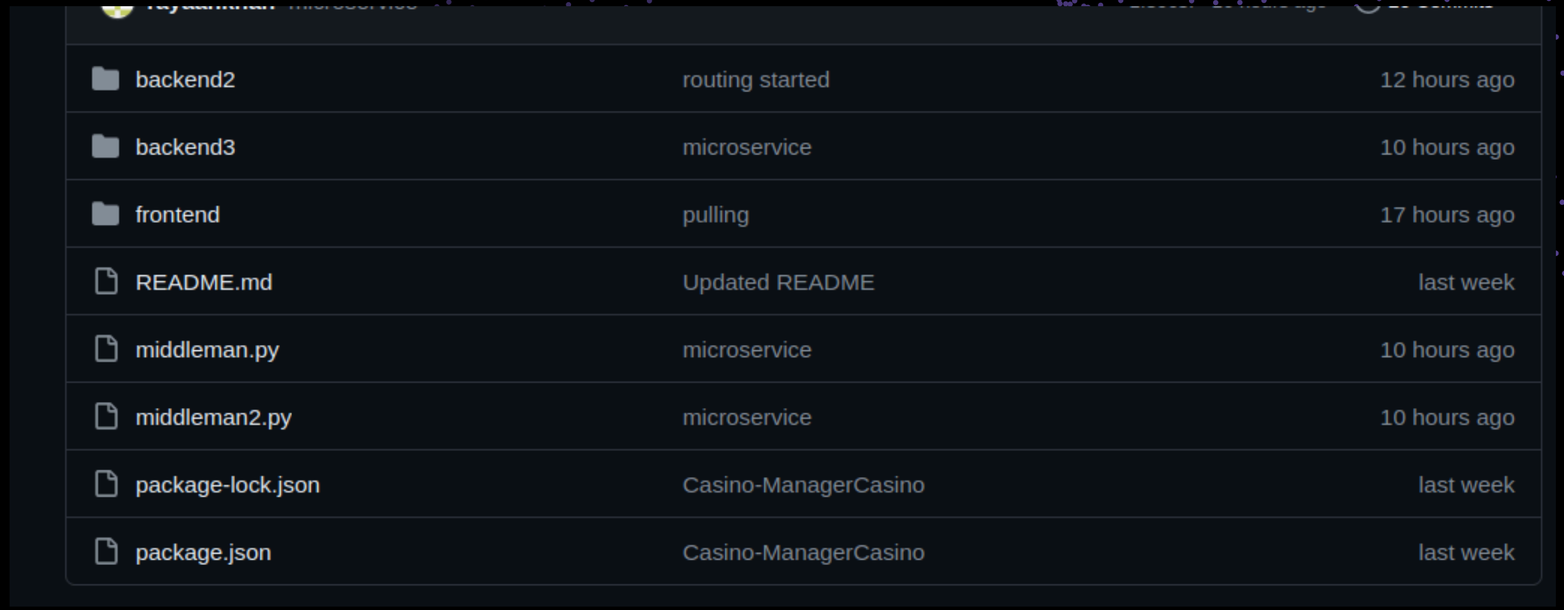
The payment microservice is running on the port 8080.

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0',port=8080)
```

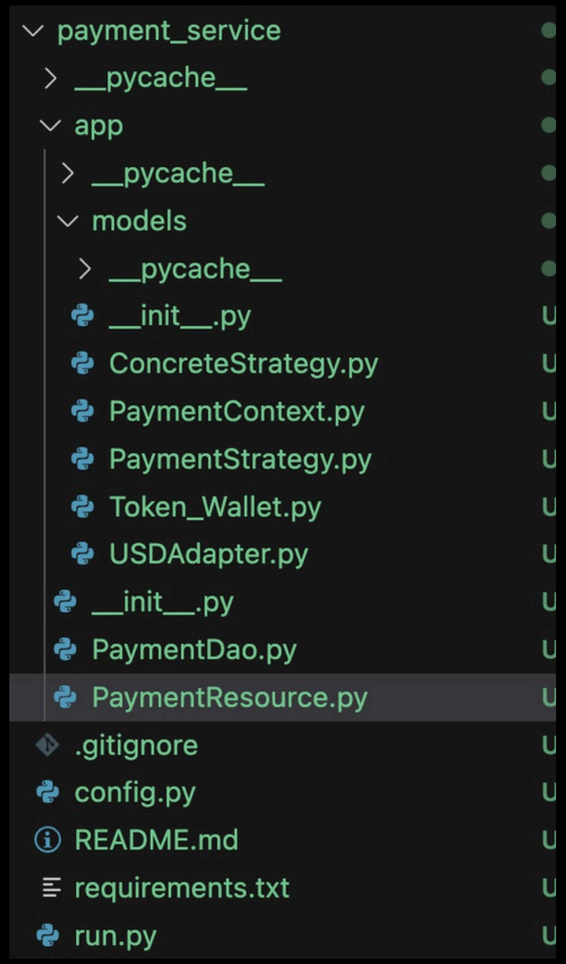
# Microservice Architecture

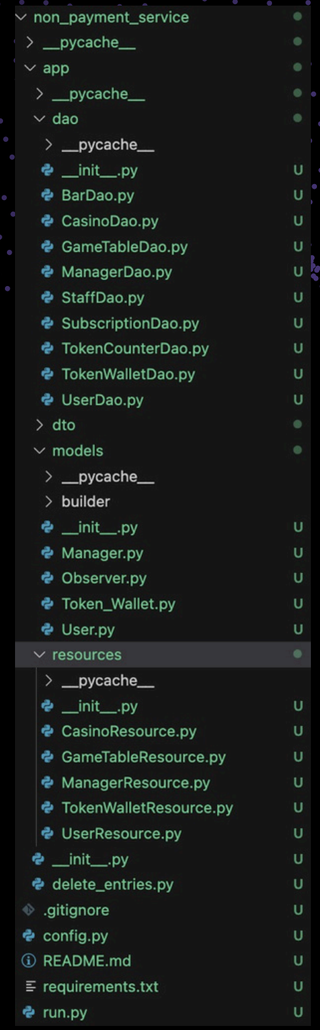
This is the directory structure



# Microservice Architecture



# 3.3 Layered  v/s Microservices Comparative Analysis

Used Locust for the performance comparison
Quality Matrix used: Response Time and Throughput

# Layered Architecture

## Observations:

1. Average response time for /users/add and /users/login is comparatively more as compared to other API calls since we are using bcrypt to hash the password and storing it.

## Aggregated Performance:

1. Overall, there were 2926 requests with no failed requests across all endpoints.
2. The median response time for all endpoints is 33 ms, while the average response time is 146.54 ms.
3. The aggregated current RPS is 60.7, indicating the total throughput of the system.

**Done 100 users ramping up at 5 users/seconds.**

### Request Statistics

| Type | Name | # Requests | # Fails | Median (ms) | Average (ms) | Current RPS |
|---|---|---|---|---|---|---|
| POST | /all_casinos | 496 | 0 | 5 | 17.67 | 10 |
| POST | /casino_info | 484 | 0 | 6 | 23.71 | 10.2 |
| POST | /users/add | 529 | 0 | 370 | 377.01 | 9.9 |
| POST | /users/login | 507 | 0 | 340 | 350.82 | 10.4 |
| POST | /wallet/addBalance | 463 | 0 | 21 | 37.29 | 10.2 |
| POST | /wallet/update | 447 | 0 | 17 | 31.23 | 10 |
| | Aggregated | 2926 | 0 | 33 | 146.54 | 60.7 |

### Response Time Statistics

| Method | Name | 50%ile (ms) | 60%ile (ms) | 70%ile (ms) | 80%ile (ms) | 90%ile (ms) | 95%ile (ms) | 99%ile (ms) | 100%ile (ms) |
|---|---|---|---|---|---|---|---|---|---|
| POST | /all_casinos | 5 | 7 | 12 | 22 | 38 | 81 | 200 | 330 |
| POST | /casino_info | 6 | 12 | 22 | 38 | 62 | 99 | 180 | 330 |
| POST | /users/add | 370 | 380 | 400 | 420 | 470 | 500 | 620 | 960 |
| POST | /users/login | 340 | 350 | 370 | 400 | 450 | 510 | 680 | 920 |
| POST | /wallet/addBalance | 21 | 27 | 34 | 48 | 73 | 130 | 210 | 380 |
| POST | /wallet/update | 17 | 22 | 27 | 33 | 67 | 92 | 220 | 460 |
| | Aggregated | 33 | 81 | 290 | 350 | 390 | 440 | 550 | 960 |

# Microservice Architecture

- **Overall Observations:**
  - User-related endpoints show higher response times, possibly due to the complexity of the operations involved, such as user authentication and account management.

- **Aggregated Performance:**
  - Overall, there were 2447 requests with no failed requests across all endpoints.
  - The median response time for all endpoints is 76 ms, while the average response time is 200.4 ms.
  - The aggregated current RPS is 58, indicating the total throughput of the system.

**Done 100 users ramping up at 5 users/seconds.**

## Request Statistics

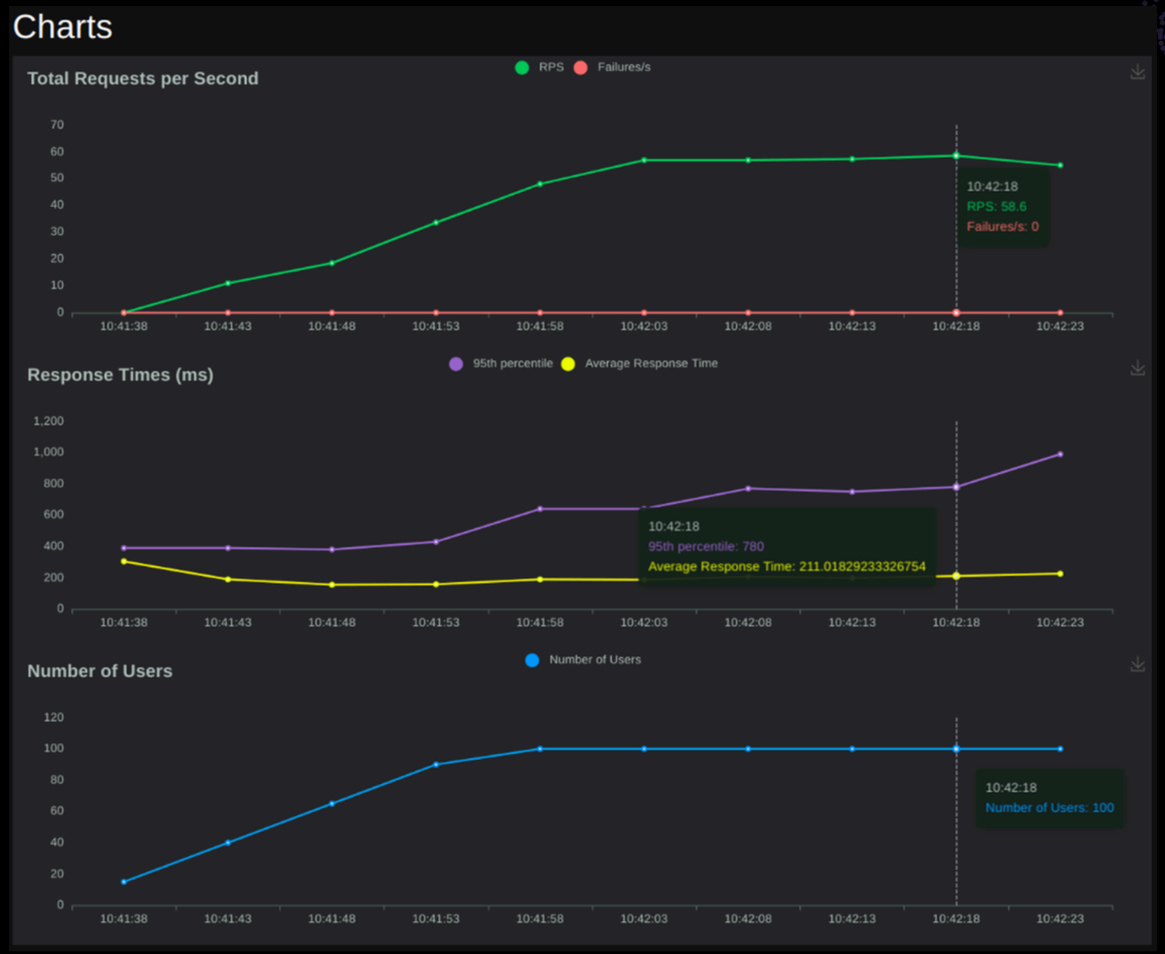
| Type | Name | # Requests | # Fails | Median (ms) | Average (ms) | Min (ms) | Max (ms) | Average size (bytes) | Current RPS | Current Failures/s |
|---|---|---|---|---|---|---|---|---|---|---|
| POST | /all_casinos | 410 | 0 | 21 | 40.01 | 3 | 830 | 392 | 9.5 | 0 |
| POST | /casino_info | 404 | 0 | 30 | 65.06 | 4 | 1180 | 570.47 | 8.9 | 0 |
| POST | /users/add | 448 | 0 | 460 | 533.81 | 230 | 1789 | 51 | 8.8 | 0 |
| POST | /users/login | 429 | 0 | 440 | 488.22 | 198 | 1288 | 606 | 9.8 | 0 |
| POST | /wallet/addRecordBalance | 386 | 0 | 52 | 91.02 | 25 | 1092 | 21 | 8.7 | 0 |
| POST | /wallet/update | 370 | 0 | 33 | 63.35 | 14 | 927 | 56 | 9.3 | 0 |
| | Aggregated | 2447 | 0 | 76 | 224.7 | 3 | 1789 | 287.22 | 55 | 0 |

## Response Time Statistics

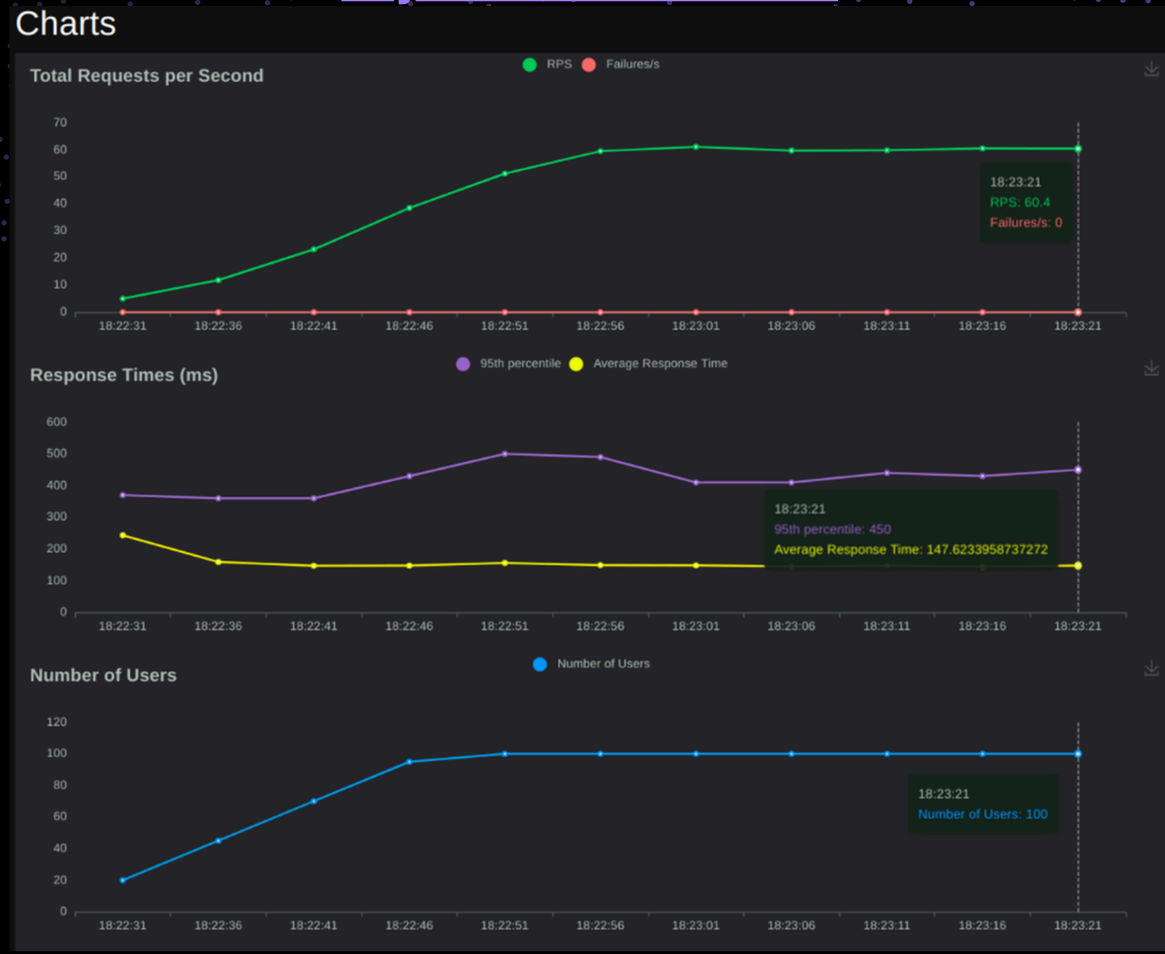
| Method | Name | 50%ile (ms) | 60%ile (ms) | 70%ile (ms) | 80%ile (ms) | 90%ile (ms) | 95%ile (ms) | 99%ile (ms) | 100%ile (ms) |
|---|---|---|---|---|---|---|---|---|---|
| POST | /all_casinos | 22 | 27 | 39 | 57 | 100 | 130 | 250 | 830 |
| POST | /casino_info | 30 | 42 | 59 | 89 | 150 | 230 | 440 | 1200 |
| POST | /users/add | 460 | 520 | 600 | 690 | 840 | 1000 | 1300 | 1800 |
| POST | /users/login | 440 | 510 | 540 | 630 | 790 | 910 | 1200 | 1300 |
| POST | /wallet/addRecordBalance | 52 | 63 | 83 | 120 | 180 | 250 | 770 | 1100 |
| POST | /wallet/update | 33 | 41 | 51 | 73 | 140 | 250 | 550 | 930 |
| | Aggregated | 76 | 190 | 340 | 440 | 600 | 770 | 1100 | 1800 |

# Microservice v/s Layered

## Microservice Architecture




## Layered Architecture



# Microservice v/s Layered

## Performance Analysis:

- Layered Architecture:
  - Offers better performance for monolithic applications:
    - Low network overhead with direct method calls between components.
    - Predictable response times under varying loads.
- Microservices Architecture:
  - Introduces network communication overhead between services:
    - Potential impact on performance, especially in latency-sensitive applications.
  - Allows for better optimization of individual services:
    - Improved overall system performance through efficient scaling and resource allocation.

# REPOSITORY

Our Github Repo link:
https://github.com/rayaankhan/Project-3-SE

The **yatharth** branch contains the code for Layered Architectural Pattern.
The **microservice** branch has the code for Microservice Architectural Pattern

# ROLES

Rayaan Khan - Builder Pattern, Gametable Play, Deletion of Casino, game table, bar, Addition of staff, modifying casinos

Yash Shivhare - Casino (Manager) & User Analytics

Yatharth Gupta - Observer Pattern, Admin implementation, Login/Registration, Layered-Microservice Analysis

Ashutosh Srivastava - Strategy Pattern, Adapter Pattern, Microservices

Manuj Garg - Builder Pattern,Microservices

# Do you have **any questions?**