Project 3 Deliverables:

Team Number: 17

Project Title: Casino Management System (CMS)

Task 1: Requirements and Subsystems

1.1 Functional Requirements:

- **User Management:**
  - Register and manage customer accounts
  - Manage employee accounts and roles
  - Track user activity and gameplay data
- **Casino Management:**
  - Manage sub-brands and individual casino information
  - Track financial data (income, expenses, profits)
  - Monitor 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.

1.2 Non-Functional Requirements:

- **Security:** Protect user data from unauthorized access
- **Performance:** Ensure fast response times and efficient operations
- **Flexibility:** Easily add new casino sub-brands and game types
- **Usability:** Provide a user-friendly interface for both customers and casinos

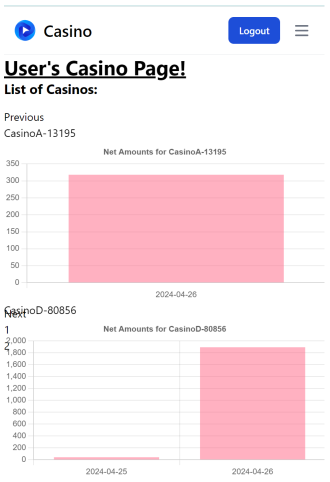
Architecturally Significant Requirements:

- **User Management:** Securely storing and managing user data is crucial for system security and privacy.
- **Flexibility/Extendibility:** The ability to add new casinos and game types easily impacts the design of the data model and system architecture.

1.3 Subsystem Overview:

- **User Management Subsystem:** Handles user registration, authentication, and authorization. The password is stored using bcrypt for safety purpose. Admin has the power to add new managers while manager can log in and perform its functions like creating casinos etc.
- **Casino Management Subsystem:** Manages casino information, financial data, and game table performance.

  - Includes creation of a Casino using a builder pattern with the no of game tables, bars, staff given as user input if you are a manager.
  - Lists all the casinos if you are a user. We can click on a particular casino and the gametables and bars present in the casino will be displayed . User can click on any gametable and play which can lead to his/her profit or loss that will be stored in the database.
- **Game Management Subsystem:** Configures and tracks game tables and gameplay data.
- **Payment Processing Subsystem:** Handles chip purchases, redemptions, and integrates with external payment gateways
  - The entire subsystem is built on Strategy pattern for different modes of payments including card, cash and upi.
  - Have a method to convert cash into tokens and these tokens are then used to play in the casino.
  - Implemented Adapter pattern for handling cash payment through US Dollars.
- **Notification Subsystem:** In this subsystem the user is able to subscribe to any casino that it wants to and the manager of that casino can notify these users for any new updates.
- **Analytic Subsystem:** In this subsystem, both the user and the manager can analyze the overall money they have spent and the overall profit/loss they are in.
  - **User:** The user can see the profits/losses he has made on the casinos he went to based on a particular day and in terms of usability, easily keep track of his overall expenses in different casinos.


Casino

Logout

User's Casino Page!

List of Casinos:

Previous

CasinoA-13195

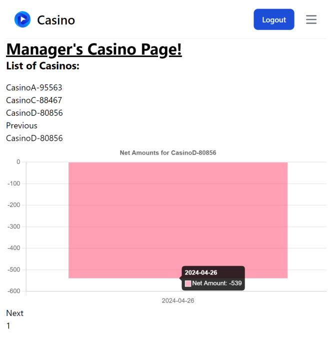
Next

CasinoD-80856

1

2

  - **Manager**:
    - A manager can analyze all the casinos under him and see the overall profit/loss he has made (on a daily basis graphically). He can analyze which casino is making more profit and what’s the daily trend of the casinos (implying the season when he expects the most profits). He can also easily check any abnormal patterns (for instance a casino going into loss drastically or consistently) and take measures.



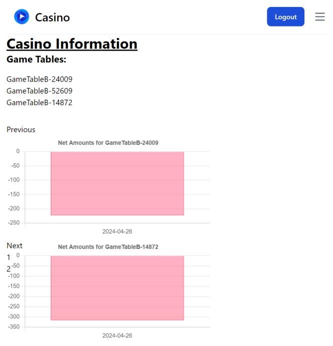
Manager's Casino Page!

List of Casinos:

CasinoA-95563
CasinoC-88467
CasinoD-80856
Previous
CasinoD-80856

Next
1

- Within every casino as well, the manager can analyze all the profits/losses made within all the game tables in that casino as well which can be used to analyze which game tables are bringing him more profits so that he can increase those particular kinds while creating a new casino.



Casino Information

Game Tables:

GameTableB-24009
GameTableB-52609
GameTableB-14872

Previous

Next
1
2

Task 2: Architecture Framework

2.1 Stakeholder Identification (IEEE 42010):

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

2.2 Major Design Decisions (ADRs):

ADR	Decision	Rationale
1	Use SQL database, 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

Task 3: Architectural Tactics and Patterns

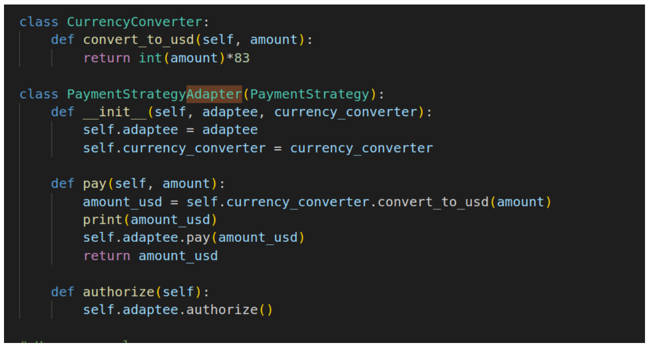
3.1 Architectural Tactics:

- **Security Tactics**
  - Authenticate Users - During login we are checking the credentials of the user and using bcrypt for the password.
  - Authorize Users - Using the token saved in the localstorage we are fetching the userid and and their roles at every crucial backend calls.
  - Data Confidentiality - Password has been encrypted using bcrypt library for confidentiality.
- **Availability Tactics**
  - Exception Handling - In order to avoid a total shutdown, the system gracefully handles faults or errors that arise from backend API calls. Rather, it ensures that the system stays functional and responsive to user requests by responding with instructive error messages. This strategy improves the system's overall availability and dependability by proactively addressing possible problems, which supports continuous service delivery.
- **Modifiability Tactics**
  - Semantic Coherence - Consistently maintaining semantic coherence throughout the codebase is key to enhancing modifiability. This ensures that the code remains clear and understandable across all components, enabling developers to easily make changes or additions without complexity. By organizing code cohesively, development becomes simpler, leading to a more adaptable architecture that can efficiently evolve to meet evolving business needs.

3.2 Implementation Patterns:

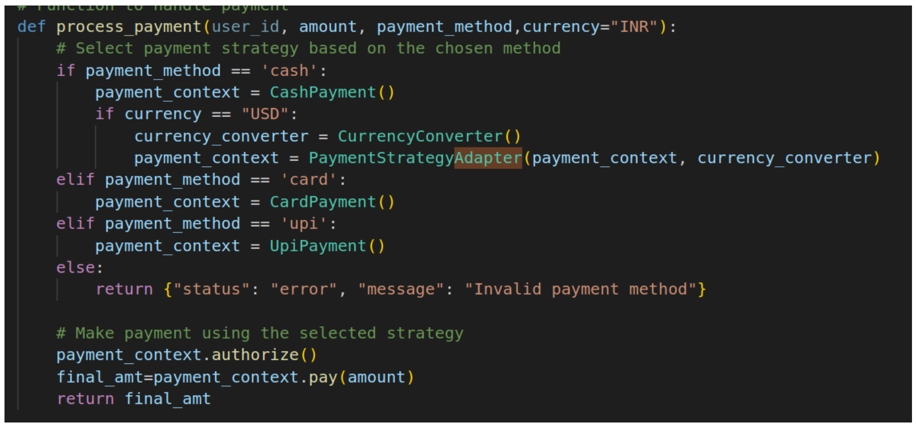
- **Adapter Pattern (Implemented):**

  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
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()
```

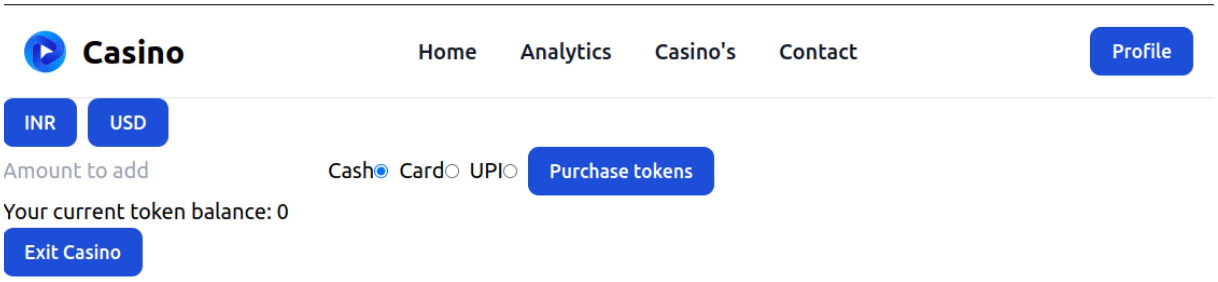
In the process_payment function if the payment method is cash we use the **CashPayment** as paymentcontext. If the currency used is USD$, we wrap the adapter **PaymentStrategyAdpater** around the payment context(**CashPayment**) . When we call pay method it directs to the method pay defined in the adapter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



```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
```

- **Strategy Pattern (Implemented):** To handle different payment methods flexibly.

  The strategy pattern was implemented between cash, card and UPI



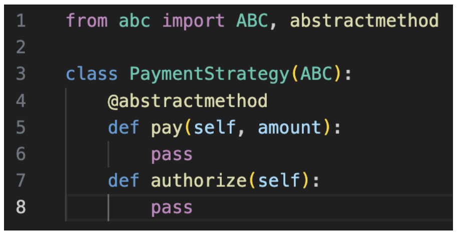
Your current token balance: 0

In our CMS, we've seamlessly integrated a robust payment strategy system that enhances user flexibility and convenience when conducting transactions within our platform. Leveraging the versatile `Strategy Pattern`, we've crafted an adaptable architecture that empowers users to seamlessly choose their preferred payment methods while ensuring scalability and maintainability of our payment system.

With this implementation, users can select from a variety of payment methods, including cash, card, and UPI, each tailored to suit their individual preferences and needs. This flexibility allows users to conduct transactions using the method that best aligns with their financial habits and requirements.

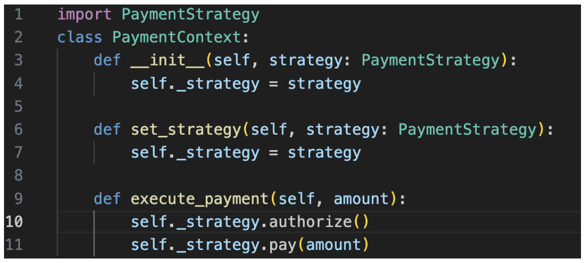
Behind the scenes, when a user selects a payment method, our system dynamically applies the corresponding payment strategy to execute the transaction seamlessly. For instance, when a user opts for a card payment, our system utilizes the Card Payment Strategy, facilitating secure and efficient card-based transactions. Similarly, for UPI payments, the UPI Payment Strategy seamlessly handles the transaction process, providing users with a hassle-free experience.

PaymentStrategy.py



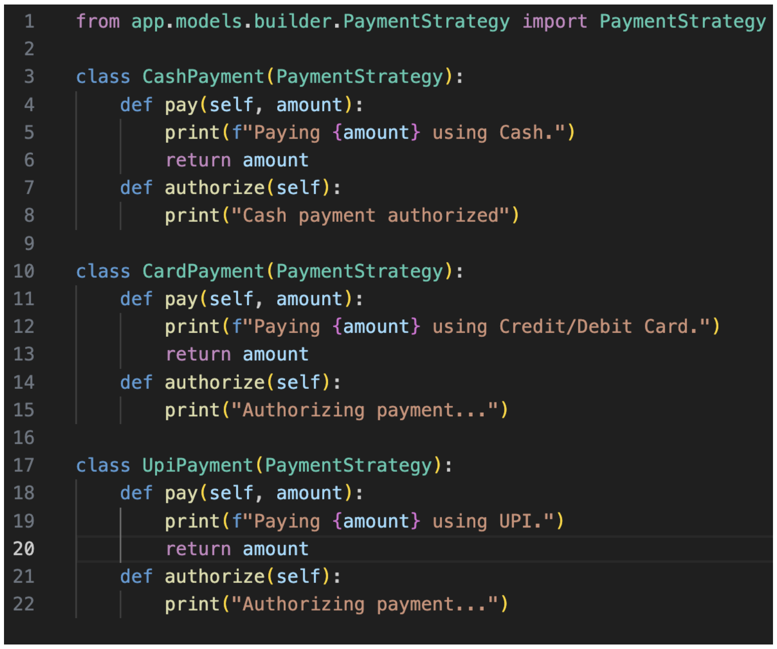
```python
from abc import ABC, abstractmethod

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

PaymentContext.py



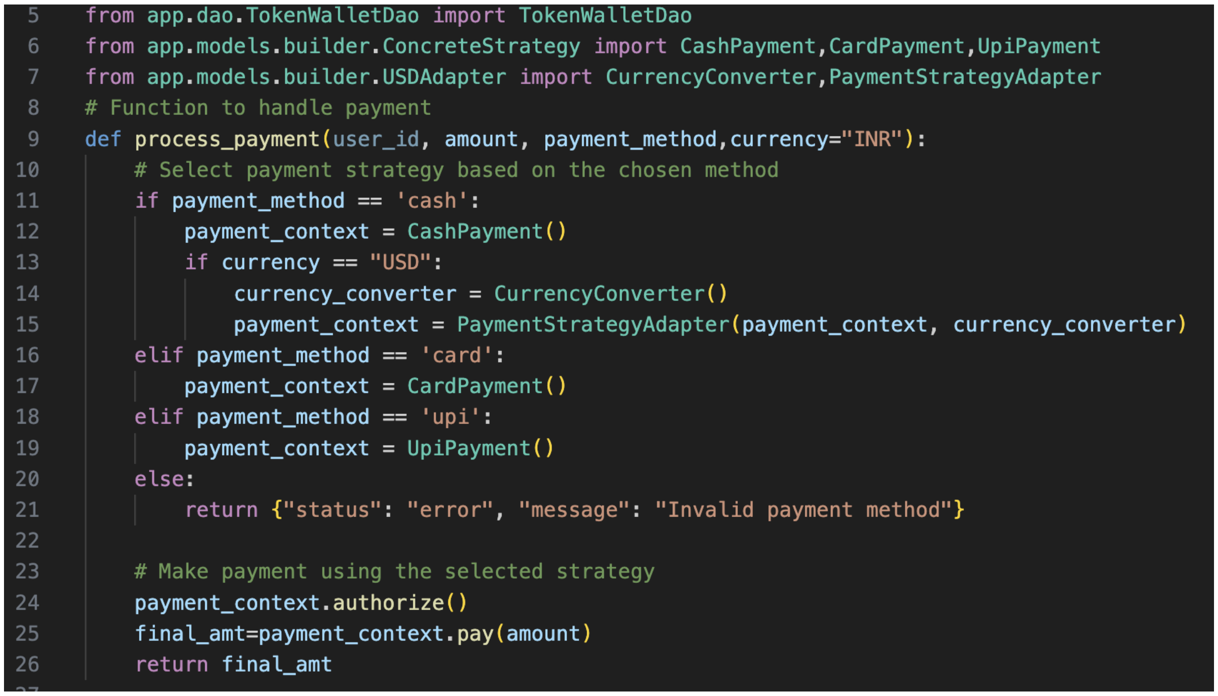
```python
import PaymentStrategy
class PaymentContext:
    def __init__(self, strategy: PaymentStrategy):
        self._strategy = strategy

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

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

ConcreteStrategy.py

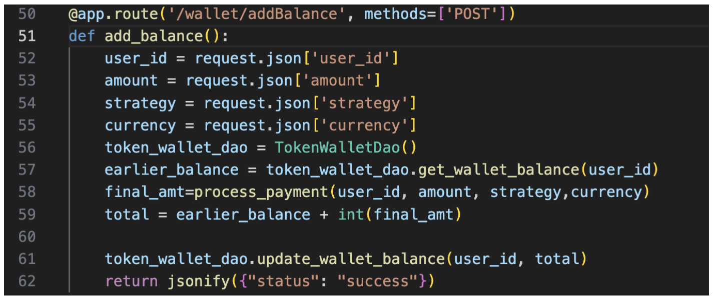


```python
from app.models.builder.PaymentStrategy import PaymentStrategy

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...")
```

TokenWalletResource.py

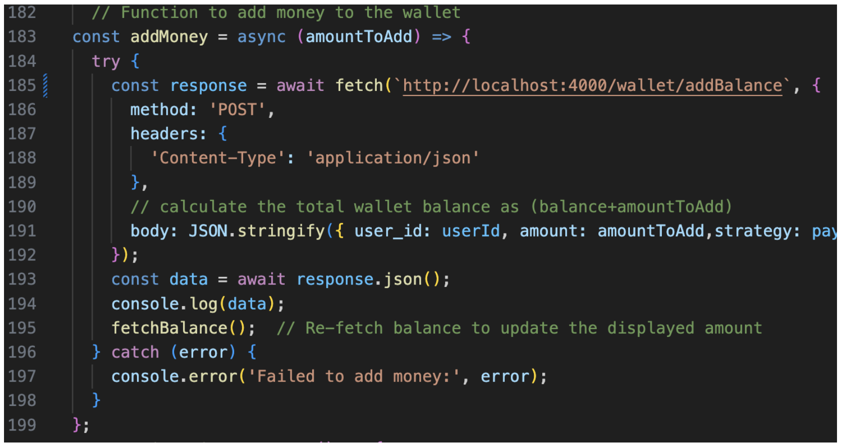


```python
from app.dao.TokenWalletDao import TokenWalletDao
from app.models.builder.ConcreteStrategy import CashPayment,CardPayment,UpiPayment
from app.models.builder.USDAdapter import CurrencyConverter,PaymentStrategyAdapter
# Function to handle payment
def process_payment(user_id, amount, payment_method,currency="INR"):
    # 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"}

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



```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"})
```

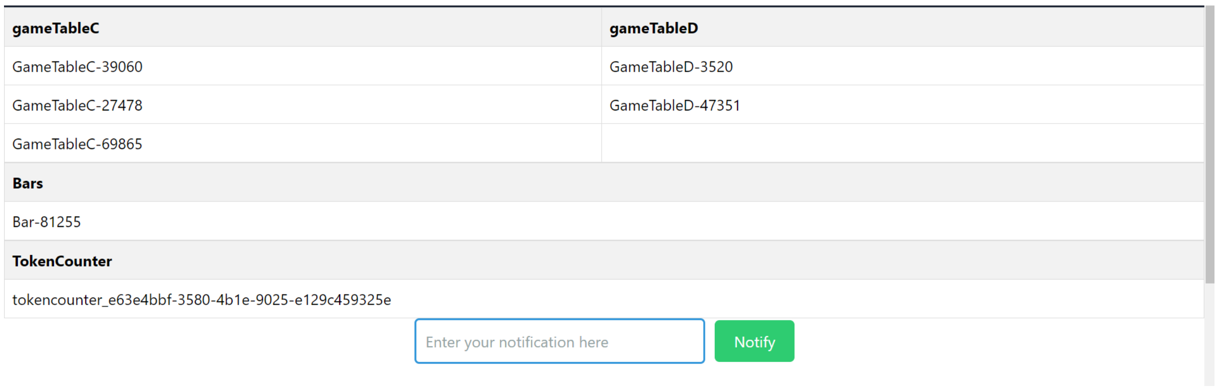
Frontend request



```javascript
  // Function to add money to the wallet
const addMoney = async (amountToAdd) => {
  try {
    const response = await fetch(`http://localhost:4000/wallet/addBalance`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      // calculate the total wallet balance as (balance+amountToAdd)
      body: JSON.stringify({ user_id: userId, amount: amountToAdd,strategy: pay
    });
    const data = await response.json();
    console.log(data);
    fetchBalance();  // Re-fetch balance to update the displayed amount
  } catch (error) {
    console.error('Failed to add money:', error);
  }
};
```

Furthermore, our payment strategy system promotes code reusability and maintainability by encapsulating payment method-specific logic within separate strategy classes. This modular approach simplifies code management and facilitates efficient testing and debugging.

- **Observer Pattern (Implemented):** For notification to subscribed users.



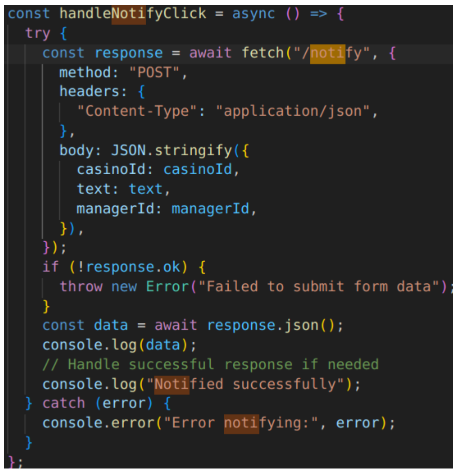
gameTableC	gameTableD
GameTableC-39060	GameTableD-3520
GameTableC-27478	GameTableD-47351
GameTableC-69865	
Bars	
Bar-81255	
TokenCounter	
tokencounter_e63e4bbf-3580-4b1e-9025-e129c459325e	

In our CMS, we have integrated a feature that allows users to subscribe to or unsubscribe from specific casinos. This subscription system allows users to receive updates and notifications about the casinos they are interested in. We used the `Observer Pattern` to implement this functionality

When a user subscribes to a casino, they will begin to receive notifications from the casino manager. These notifications can include updates about new game tables, changes in staff, or special events and promotions. This feature enhances the user experience by keeping them connected and informed about their favorite casinos.

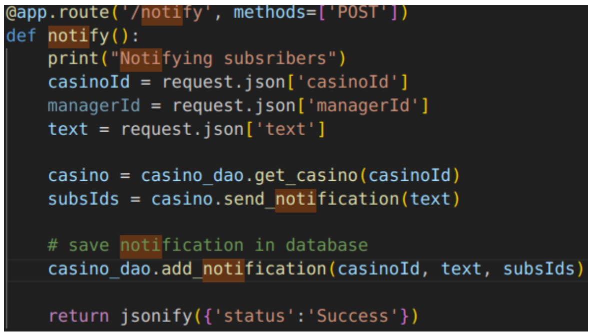
Managers have the ability to send notifications to all subscribed users. This can be done through the manager's interface. By clicking the "Notify" button, managers can send a message to all subscribers of their casino. This feature allows managers to effectively communicate updates and important information to their casino's user base.

The process of sending notifications is handled through the `/notify` API. When a manager sends a notification, the backend code retrieves a list of all users subscribed to the casino and then sends the notification to each of these users. The notification is then stored in the database and can be viewed by the user at any time on their "Notifications" page.

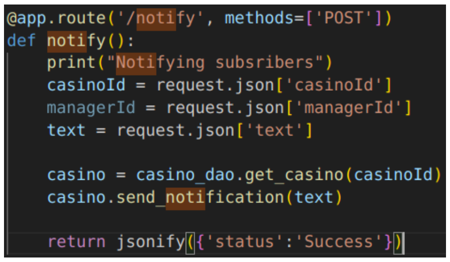


```javascript
const handleNotifyClick = async () => {
  try {
    const response = await fetch("/notify", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        casinoId: casinoId,
        text: text,
        managerId: managerId,
      }),
    });
    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

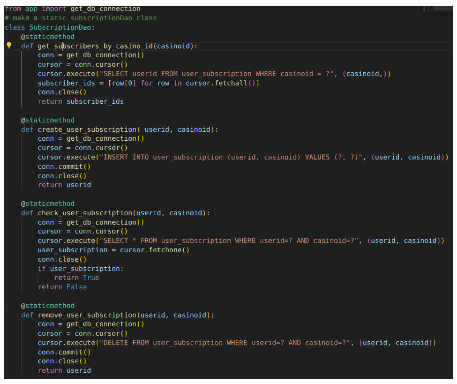


```python
@app.route('/notify', methods=['POST'])
def notify():
    print("Notifying subsribers")
    casinoId = request.json['casinoId']
    managerId = request.json['managerId']
    text = request.json['text']

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

    # save notification in database
    casino_dao.add_notification(casinoId, text, subsIds)

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



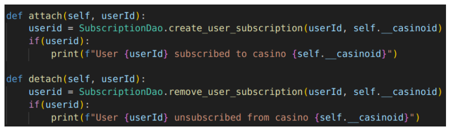
```python
@app.route('/notify', methods=['POST'])
def notify():
    print("Notifying subsribers")
    casinoId = request.json['casinoId']
    managerId = request.json['managerId']
    text = request.json['text']

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

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

backend receives the call in CasinoResource.py



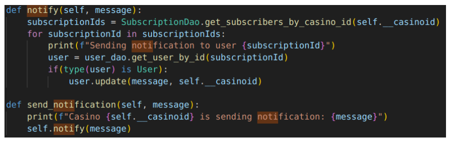
```python
from app import get_db_connection
# make a static subscriptionDao class
class SubscriptionDao:
    @staticmethod
    def get_subscribers_by_casino_id(casinoid):
        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT userid FROM user_subscription WHERE 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



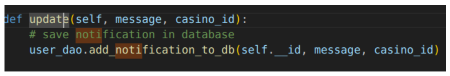
```python
def attach(self, userId):
    userid = SubscriptionDao.create_user_subscription(userId, self.__casinoid)
    if(userid):
        print(f"User {userId} subscribed to casino {self.__casinoid}")

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

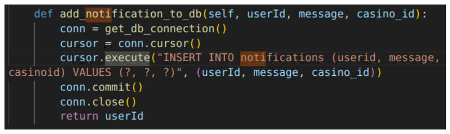


```python
def notify(self, message):
    subscriptionIds = SubscriptionDao.get_subscribers_by_casino_id(self.__casinoid)
    for subscriptionId in subscriptionIds:
        print(f"Sending notification to user {subscriptionId}")
        user = user_dao.get_user_by_id(subscriptionId)
        if(type(user)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



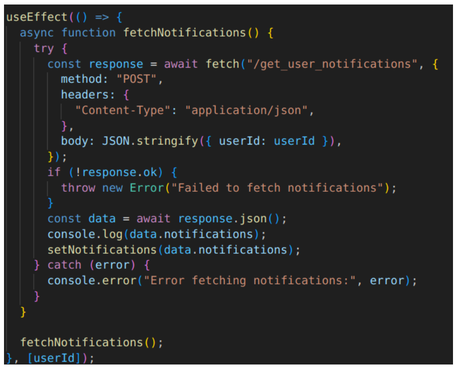
```python
def update(self, message, casino_id):
    # save notification in database
    user_dao.add_notification_to_db(self.__id, message, casino_id)
```



```python
    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
```

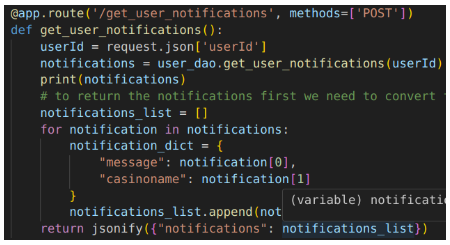
adds the notification to db.

User notifications are fetched on the `/get_user_notifications` API, which uses the get_user_notifications function of the user_dao, converts it to a notifs array, and returns to the frontend. The frontend is shown in the image above.



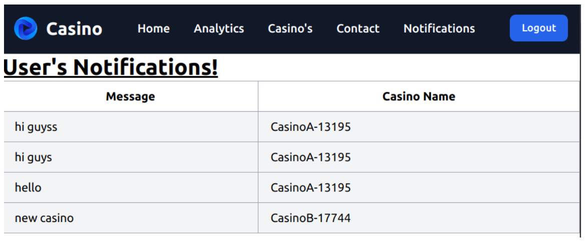
```javascript
useEffect(() => {
  async function fetchNotifications() {
    try {
      const response = await fetch("/get_user_notifications", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({ userId: userId }),
      });
      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



```python
@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]
        }                                    (variable) notificati
        notifications_list.append(not
    return jsonify({"notifications": notifications_list})
```

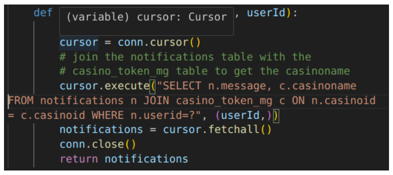
backend receives the request in UserResource.py



User's Notifications!

Message	Casino Name
hi guyss	CasinoA-13195
hi guys	CasinoA-13195
hello	CasinoA-13195
new casino	CasinoB-17744

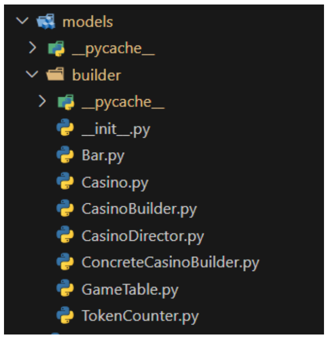
User interface for viewing notifications



```python
    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
```

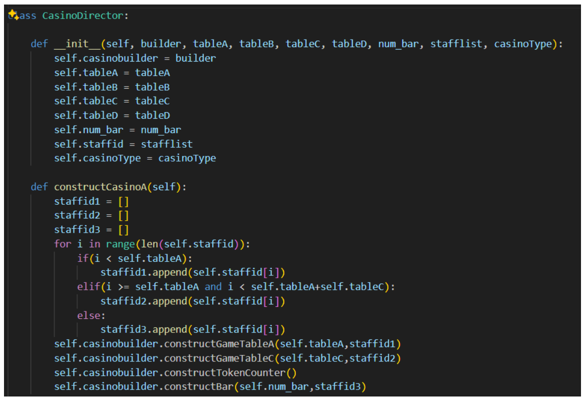
- **Builder Pattern (Implemented):**

  We have implemented Builder Pattern for creation of casino. Below is the directory structure which is in the lines of Builder Pattern that contains a Director to control the flow of creation , an interface which is implemented by different concrete builders (they have different implementations of the same function defined in the interface).



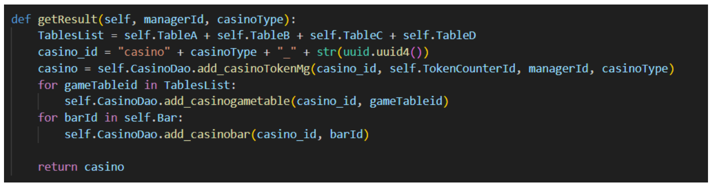
```
models
  __pycache__
  builder
    __pycache__
    __init__.py
    Bar.py
    Casino.py
    CasinoBuilder.py
    CasinoDirector.py
    ConcreteCasinoBuilder.py
    GameTable.py
    TokenCounter.py
```

In the **CasinoDirector.py** implementation of different types of casinos (there are 4 types of casinos) are present which builds the casinos according to pre defined conditions.



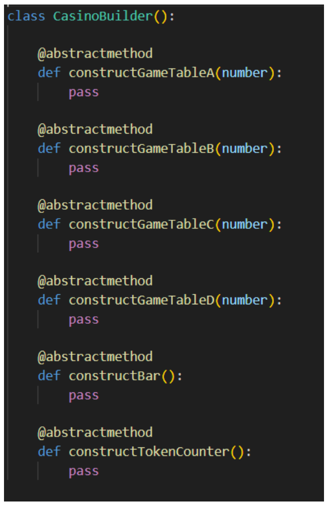
```python
ass CasinoDirector:

    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)
```

Below is the getResult function to return the casino built that is required by the user



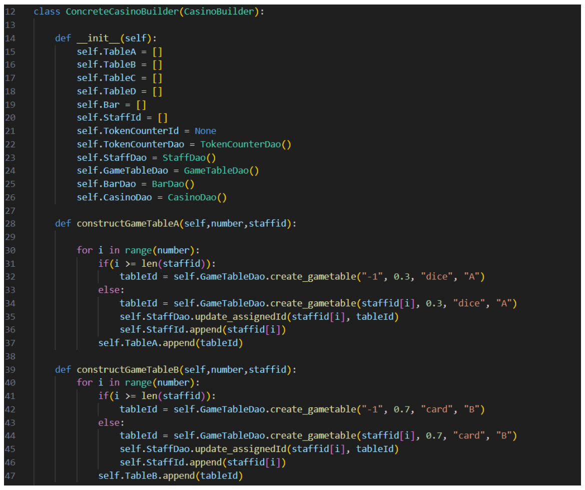
```python
def getResult(self, managerId, casinoType):
    TablesList = self.TableA + self.TableB + self.TableC + self.TableD
    casino_id = "casino" + casinoType + "_" + str(uuid.uuid4())
    casino = self.CasinoDao.add_casinoTokenMg(casino_id, self.TokenCounterId, managerId, casinoType)
    for gameTableid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

    return casino
```

CasinoBuilder.py file defines the functions that should be implemented by concrete builders.(here we have only one builder but multiple builders can be made and they should inherit the common interface).



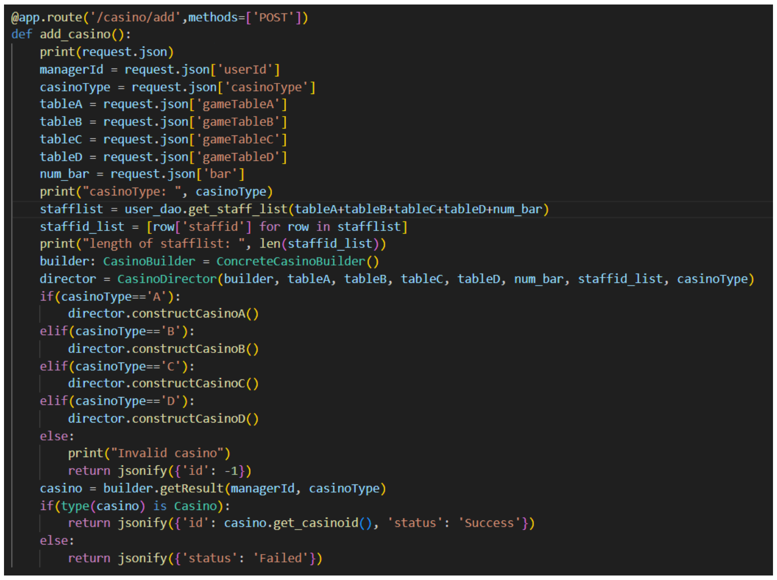
```python
class CasinoBuilder():

    @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
```

ConcreteCasinoBuilder.py inherits the CasinoBuilder class where all the implementations of gametable, bar, tokencounter creation is present.



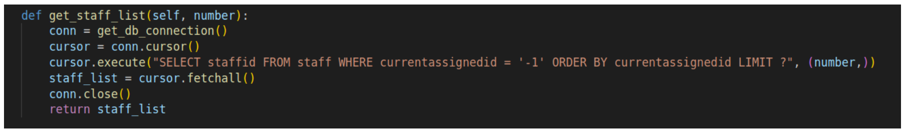
```python
class ConcreteCasinoBuilder(CasinoBuilder):

    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)
```

CasinoResource.py contains function add_casino which takes the request at the frontedd path '/casino/add' and creates a casino using Builder pattern.This is done by retrieving the no of different gametables the user wants and first assigning staff members to them . A director is then made which handles the construction of the casino and finally we return the CasinoId.



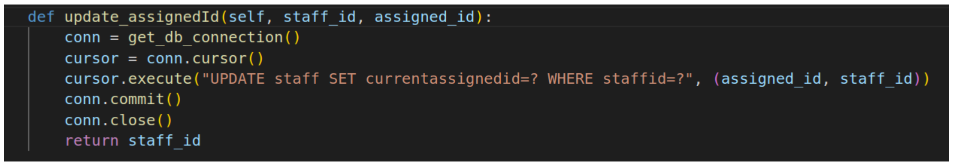
```
@app.route('/casino/add',methods=['POST'])
def add_casino():
    print(request.json)
    managerId = request.json['userId']
    casinoType = request.json['casinoType']
    tableA = request.json['gameTableA']
    tableB = request.json['gameTableB']
    tableC = request.json['gameTableC']
    tableD = 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'})
```

This returns the staffid(s) for the number of gametables that the user wants .



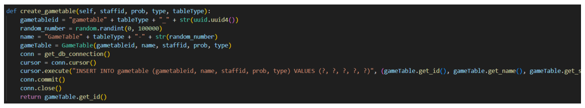
```
def get_staff_list(self, number):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT staffid FROM staff WHERE currentassignedid = '-1' ORDER BY currentassignedid LIMIT ?", (number,))
    staff_list = cursor.fetchall()
    conn.close()
    return staff_list
```

This function is defined in **StaffDao.py** which updates the id of the staff so to assign it to a particular gametable.



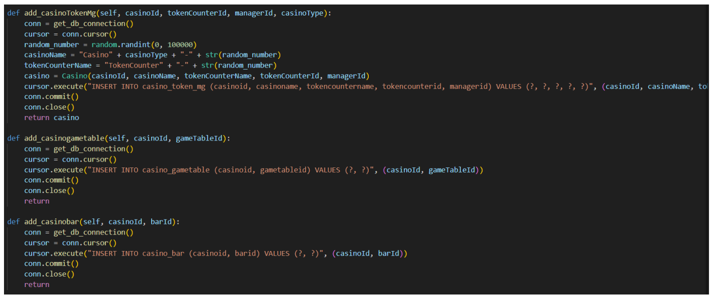
```
def update_assignedId(self, staff_id, assigned_id):
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("UPDATE staff SET currentassignedid=? WHERE staffid=?", (assigned_id, staff_id))
    conn.commit()
    conn.close()
    return staff_id
```

This function is defined in **GameTableDao.py** which inserts an entry of a gametable into the gametable Table



```
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()
```

These functions are defined in **CasinoDao.py** which adds a token counter and a bar into the corresponding tables.



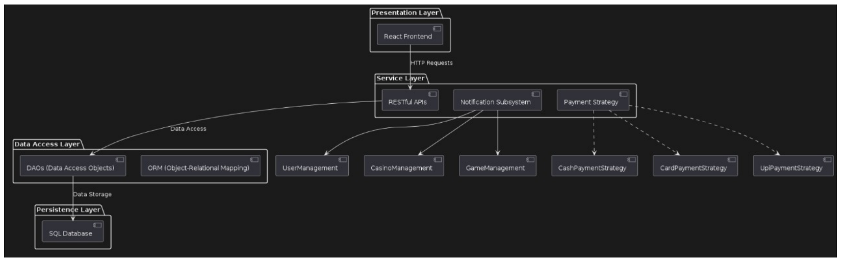
```
def add_casinoTokenMg(self, casinoId, tokenCounterId, managerId, casinoType):
    conn = get_db_connection()
    cursor = conn.cursor()
    random_number = random.randint(0, 100000)
    casinoName = "Casino" + casinoType + "-" + str(random_number)
    tokenCounterName = "TokenCounter" + "-" + str(random_number)
    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
```

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

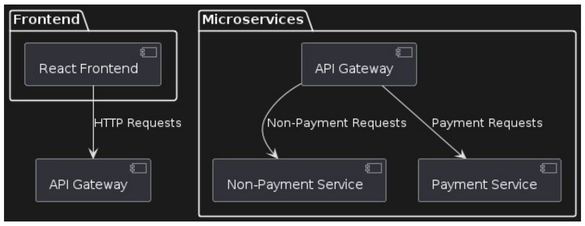
1. A frontend path is defined where we retrieve the user input.
2. We get the stafflist which doesn’t have any gametable or bar assigned to them.
3. We create a CasinoBuilder and pass it to the CasinoDirector.
4. Then we call the corresponding `constructCasino{type}()` function which is implemented in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

Diagrams:

Layered Architectural Pattern



MicroService Architectural Pattern



Task 4: Prototype Implementation and Analysis

4.1 Prototype Development:

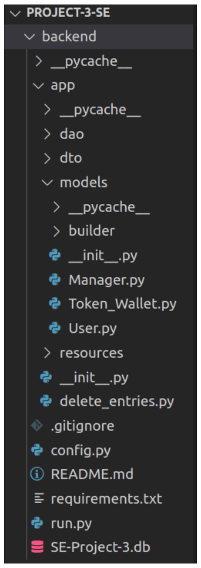
- We developed a prototype for the CMS whose repo link has been provided at the end of the document.

4.2 Architecture Analysis:

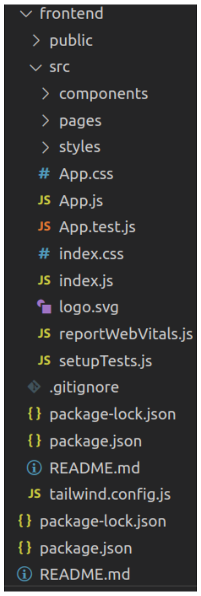
- **Layered Architecture:** Divides the system into presentation, service, data access, and persistence layers for clear separation of concerns. The directory structure for layered architectural pattern is given below:

  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)



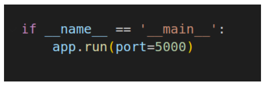
```
PROJECT-3-SE
  backend
    __pycache__
    app
      __pycache__
      dao
      dto
      models
        __pycache__
        builder
        __init__.py
        Manager.py
        Token_Wallet.py
        User.py
      resources
      __init__.py
      delete_entries.py
    .gitignore
    config.py
    README.md
    requirements.txt
    run.py
    SE-Project-3.db
```



```
frontend
  public
  src
    components
    pages
    styles
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
    setupTests.js
  .gitignore
  package-lock.json
  package.json
  README.md
  tailwind.config.js
package-lock.json
package.json
README.md
```

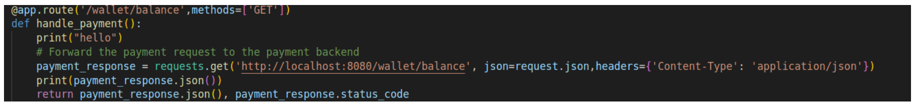
- **Microservices Architecture:** Architectural approach to build software applications as a collection of small, independent services, each focused on a specific business function and communicating via APIs.

  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.



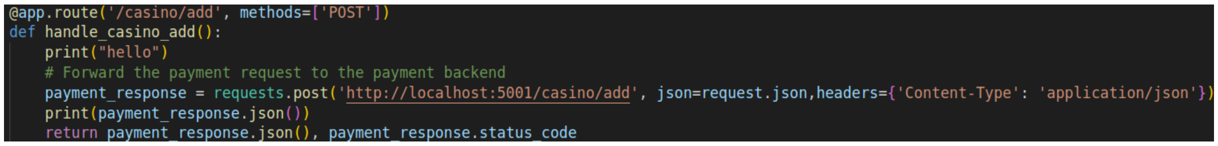
```
if __name__ == '__main__':
    app.run(port=5000)
```

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.



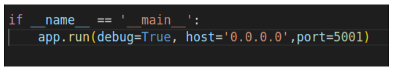
```
@app.route('/wallet/balance',methods=['GET'])
def handle_payment():
    print("hello")
    # Forward the payment request to the payment backend
    payment_response = requests.get('http://localhost:8080/wallet/balance', json=request.json,headers={'Content-Type': 'application/json'})
    print(payment_response.json())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



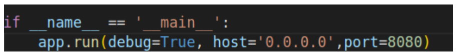
```
@app.route('/casino/add', methods=['POST'])
def handle_casino_add():
    print("hello")
    # Forward the payment request to the payment backend
    payment_response = requests.post('http://localhost:5001/casino/add', json=request.json,headers={'Content-Type': 'application/json'})
    print(payment_response.json())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

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



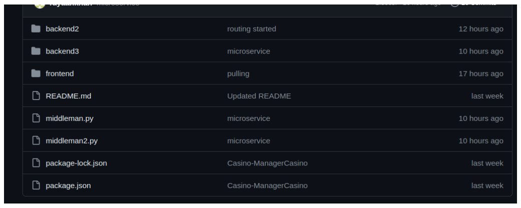
```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0',port=5001)
```

The payment microservice is running on the port 8080.

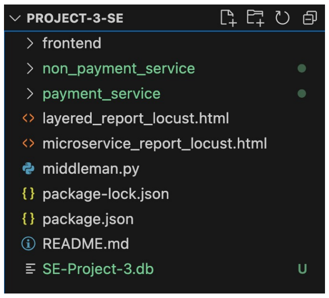


```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0',port=8080)
```

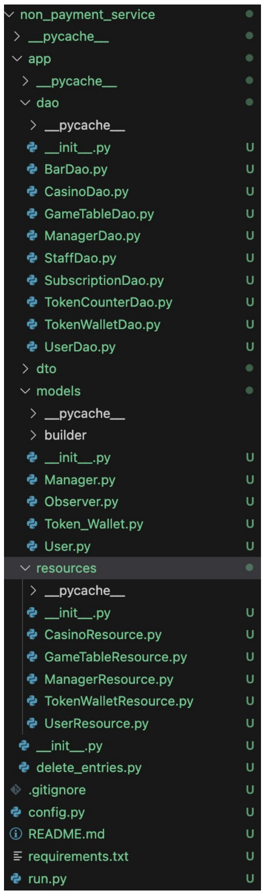
This is the directory structure



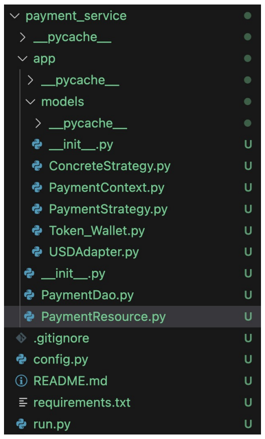
```
backend2            routing started          12 hours ago
backend3            microservice             10 hours ago
frontend            pulling                  17 hours ago
README.md           Updated README           last week
middleman.py        microservice             10 hours ago
middleman2.py       microservice             10 hours ago
package-lock.json   Casino-ManagerCasino     last week
package.json        Casino-ManagerCasino     last week
```



```
PROJECT-3-SE
  > frontend
  > non_payment_service
  > payment_service
    layered_report_locust.html
    microservice_report_locust.html
    middleman.py
    package-lock.json
    package.json
    README.md
    SE-Project-3.db                   U
```



```
non_payment_service
  > __pycache__
  v app
    > __pycache__
    v dao
      > __pycache__
        __init__.py                U
        BarDao.py                  U
        CasinoDao.py               U
        GameTableDao.py            U
        ManagerDao.py              U
        StaffDao.py                U
        SubscriptionDao.py         U
        TokenCounterDao.py         U
        TokenWalletDao.py          U
        UserDao.py                 U
    > dto
    v models
      > __pycache__
      > builder
        __init__.py                U
        Manager.py                 U
        Observer.py                U
        Token_Wallet.py            U
        User.py                    U
    v resources
      > __pycache__
        __init__.py                U
        CasinoResource.py          U
        GameTableResource.py       U
        ManagerResource.py         U
        TokenWalletResource.py     U
        UserResource.py            U
      __init__.py                  U
      delete_entries.py            U
    .gitignore                     U
    config.py                      U
    README.md                      U
    requirements.txt               U
    run.py                         U
```



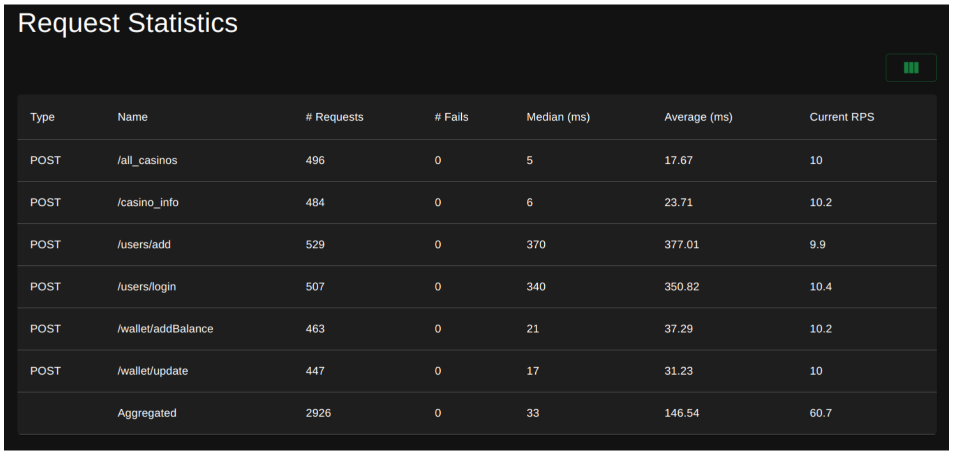
```
v payment_service
  > __pycache__
  v app
    > __pycache__
    v models
      > __pycache__
        __init__.py            U
        ConcreteStrategy.py    U
        PaymentContext.py      U
        PaymentStrategy.py     U
        Token_Wallet.py        U
        USDAdapter.py          U
      __init__.py              U
      PaymentDao.py            U
      PaymentResource.py       U
    .gitignore                 U
    config.py                  U
    README.md                  U
    requirements.txt           U
    run.py                     U
```

Comparison between layered and microservice architecture:

We used Locust for the performance comparison.

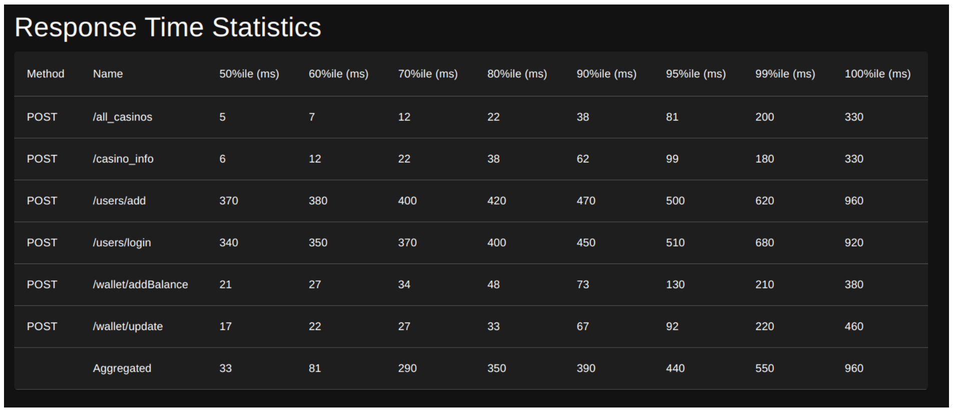
Quality Matrix used: Response Time and Throughput

Layered Architectural Pattern



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

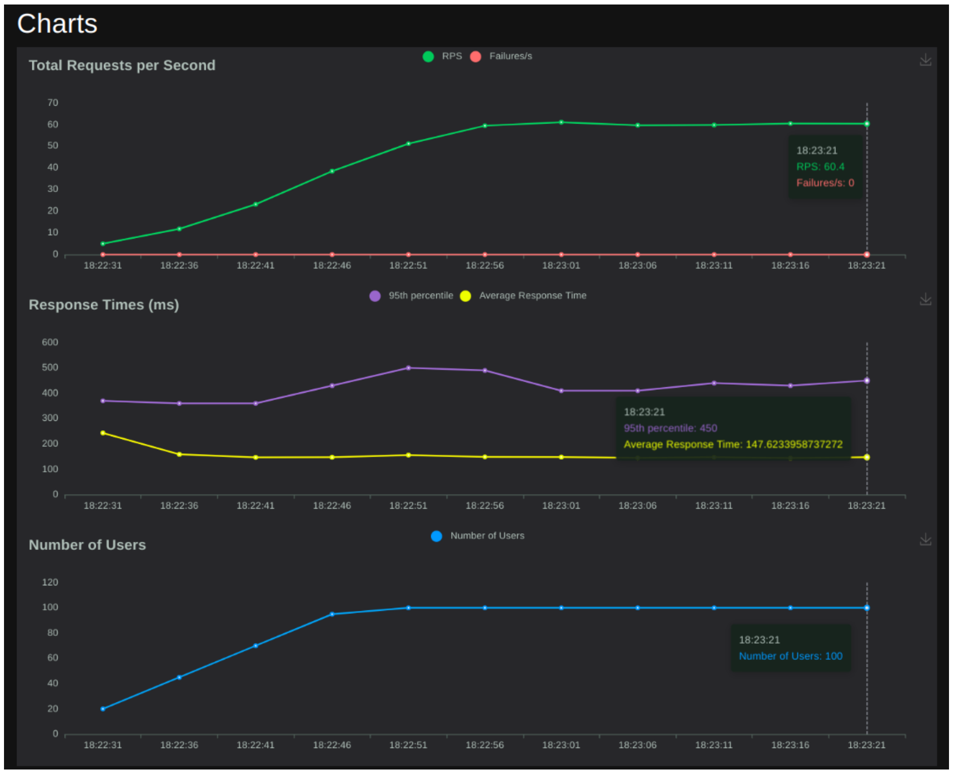
We did for 100 users ramping up at 5 users/seconds.

Observations:

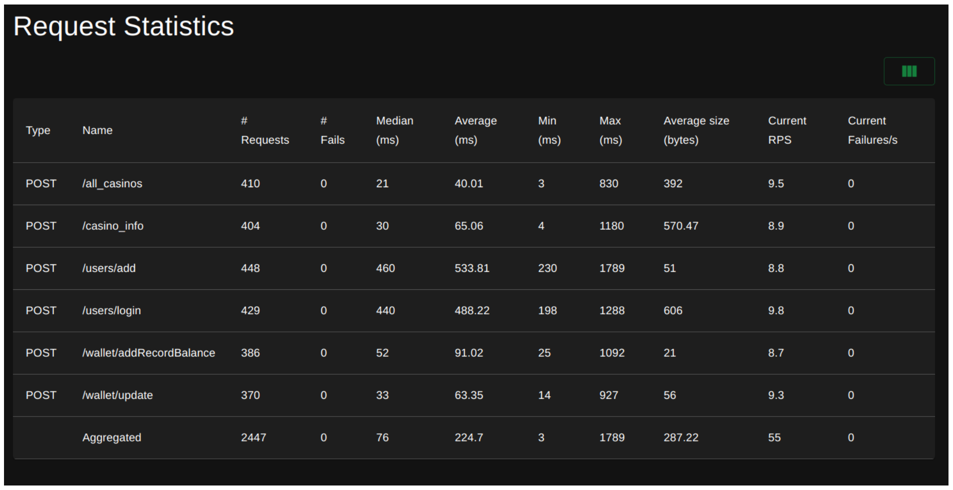
1. Average response time for /users/add and /users/login is comparatively more as compared to other API calls since we are using bcrypt to hash the password and storing it.
2. The endpoints related to casino information (/all_casinos, /casino_info) exhibit relatively low response times and stable performance.
3. The endpoints related to wallet operations (/wallet/addBalance, /wallet/update) show moderate response times, but they are relatively stable with no failed requests.

Aggregated Performance:

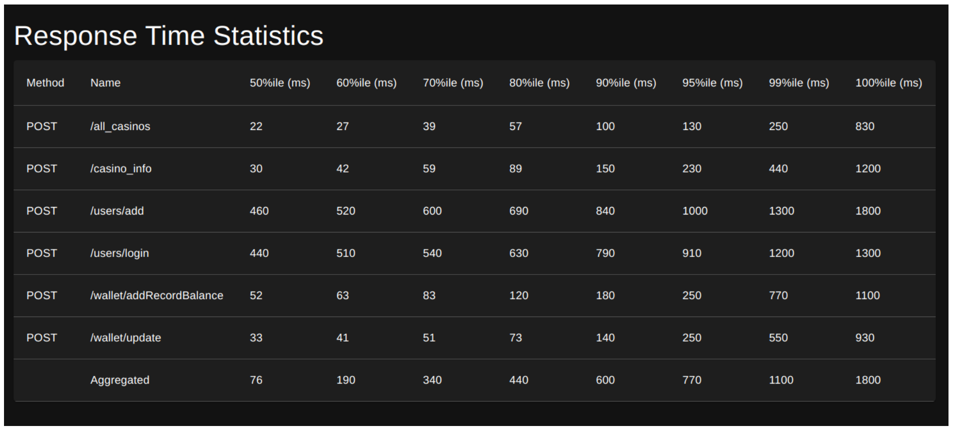
1. Overall, there were 2926 requests with no failed requests across all endpoints.
2. The median response time for all endpoints is 33 ms, while the average response time is 146.54 ms.
3. The aggregated current RPS is 60.7, indicating the total throughput of the system.



Microservice Architectural Pattern

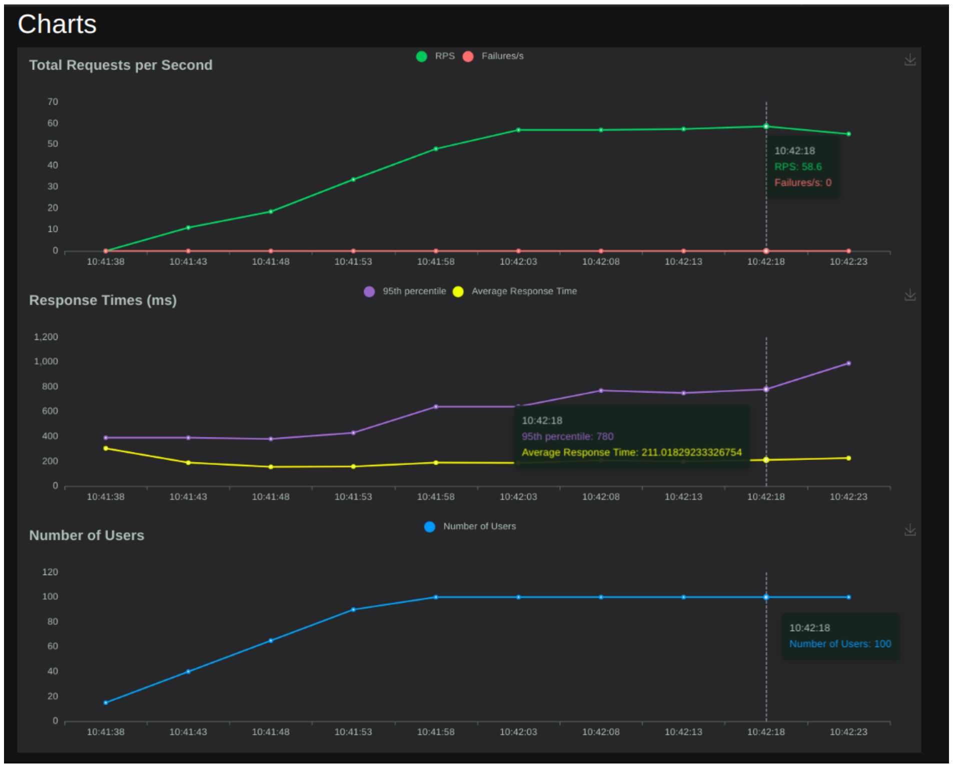

Request Statistics

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

1. **Response Time Statistics**:
   - The response time statistics provide insights into the distribution of response times across percentiles for each endpoint.
   - Endpoints related to user management (/users/add, /users/login) exhibit significantly higher response times compared to other endpoints, indicating potential performance issues or complexity in user-related operations.
   - Endpoints related to casino information (/all_casinos, /casino_info) and wallet operations (/wallet/addRecordBalance, /wallet/update) show relatively lower to moderate response times, suggesting efficient performance for these operations.
2. **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.
   - Endpoints related to casino information and wallet operations exhibit relatively lower to moderate response times, indicating efficient performance for these functionalities.
   - The aggregated response time statistics provide an overview of the overall performance of the application, with response times ranging from low to high across different percentiles.



- **IN-DEPTH ANALYSIS:**

  Response Times Comparison

  Graphical Analysis:

  1. **Microservice Architecture**:
     - The response times varied with increased user load, starting from an average response time of 154.9 ms for 65 users and escalating to 225.7 ms for 100 users.
     - The 95th percentile response times showed a marked increase from 390 ms to 990 ms as the load increased from 15 to 100 users, demonstrating how response times degrade under heavy load.
  2. **Layered Architecture**:
     - Started with a slightly higher baseline response time of 190 ms for a smaller user set and increased to 240 ms for the full load.
     - The 95th percentile response times were consistently lower than those in the microservice setup, starting at 410 ms and increasing to 800 ms, indicating a less sharp rise under stress.

  Analysis:

  - The microservice architecture exhibits a faster response under lower loads but deteriorates more significantly under high traffic, which could indicate network latency or service bottlenecks.
  - The layered architecture maintains more consistent response times, potentially due to a more predictable in-process communication setup that avoids network delays.

Throughput Metrics

Graphical Analysis:

1. **Microservice Architecture**:
    - Throughput showed an impressive increase with initial user load, peaking early before plateauing as the system became saturated.
2. **Layered Architecture**:
    - Exhibited a gradual throughput increase, which did not peak as sharply but maintained a steady increment pace throughout the test.

Analysis:

- The microservice architecture can handle a high number of transactions up to a point, beyond which it stabilizes, possibly due to reaching service limits or network constraints.
- The layered architecture ramps up more evenly, possibly reflecting a limit in resource allocation that spreads more evenly across the system components.

1. **Complexity:**
    - Layered Architecture: Has fewer components and a simpler structure, making it easier to understand and manage.
    - Microservices Architecture: Higher complexity due to the distributed nature of services, network communication overhead, and the need for additional infrastructure for service discovery, monitoring, and management.
2. **Scalability:**
    - Layered Architecture: Scaling is more challenging as the entire application is typically scaled together, limiting the ability to independently scale individual components.
    - Microservices Architecture: Better scalability as individual services can be scaled independently based on demand, allowing for more efficient resource utilization and horizontal scaling.
3. **Maintainability:**
    - Layered Architecture: Changes and updates may be easier to implement as the components are more tightly coupled, but there's a risk of unintended consequences due to dependencies.
    - Microservices Architecture: Better maintainability as services are loosely coupled, allowing for independent development, deployment, and updates without affecting other parts of the system. However, managing a larger number of services can increase operational overhead.
4. **Performance:**
    - Layered Architecture: Offers better performance for monolithic applications with low network overhead and direct method calls between components.
    - Microservices Architecture: Introduces network communication overhead between services, which can impact performance, especially in latency-sensitive applications. However, it allows for better optimization of individual services and can lead to improved overall system performance through efficient scaling and resource allocation.

Assumptions:

1. 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
2. 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.

3. The user is supposed to Exit casino while moving out of the casino so that he can cash out the money and get it back.

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