BAHRIA UNIVERSITY (KARACHI CAMPUS)

Open Ended Lab 2 – SPRING SEMESTER – 2023

(Software Design and Architecture - SEL457)

Class: **BSE 4B** (Morning)

Course Instructor: **Engr Majid Kaleem**

Lab Instructor: **Engr Muhammad Rehan Baig** **Max Marks: 6**

Student's Name: **Muhammad Shoaib Akhter Qadri** Reg. No: 79290

Note: Note: Probability of similarity is 0% copied and similar solutions will be marked as zero

SCENERIO:

StoreJinnie an online store after implementation of a system to manage their inventory, orders, and shipping processes in PHASEI. Now wanted to expand their application and also wants to add automation in their system. Multiple modules will be created using different tech stack (i.e.:Programming Languages) as separate projects because multiple technical teams working on modules to rapid application development.

Following are the main Modules of this system.

- **Payments Gateways**: Multiple Payment Gateways can be integrated into the system (Stripe, PayPal, Remitly, Easypaisa, etc.). if one payment gateway fails to process payments then the process automatically transfers to another payment gateway based oncountries.
- **Notifications:** Based on user visits and purchases from the system Newsletter Emails,Text and In App Notifications will be scheduled and dispatched to Users.
- **Message Queueing:** The system can handle a high volume of orders without overwhelming the processing capacity, as the orders are queued and processedasynchronously.
- **Affiliate Program**: This Module will provide Affiliate related portals and marketing datato Affiliates.
- **Extensions:** This Module will provide Exposed Endpoints for Integration with otherApplications and using of system public data to third party apps.

Q1 Implement best Suited Architecture for this application.
Q2 Implement best Suited Architectural Patterns and Design Patterns for this Application.

NOTE: Please solve the above scenario with explanations.

Answer

I am building **E-commerce Application** in which different Modules are including:

- **SignIn / SignUp**
- **Inventory**
- **Orders**
- **Shipping Process**
- **Add to Cart Functionality**
- **Payments Gateways**
- **Notifications**
- **Message Queueing**
- **Affiliate Program**
- **Extensions**

Here are the explanation of all modules with source code and output:

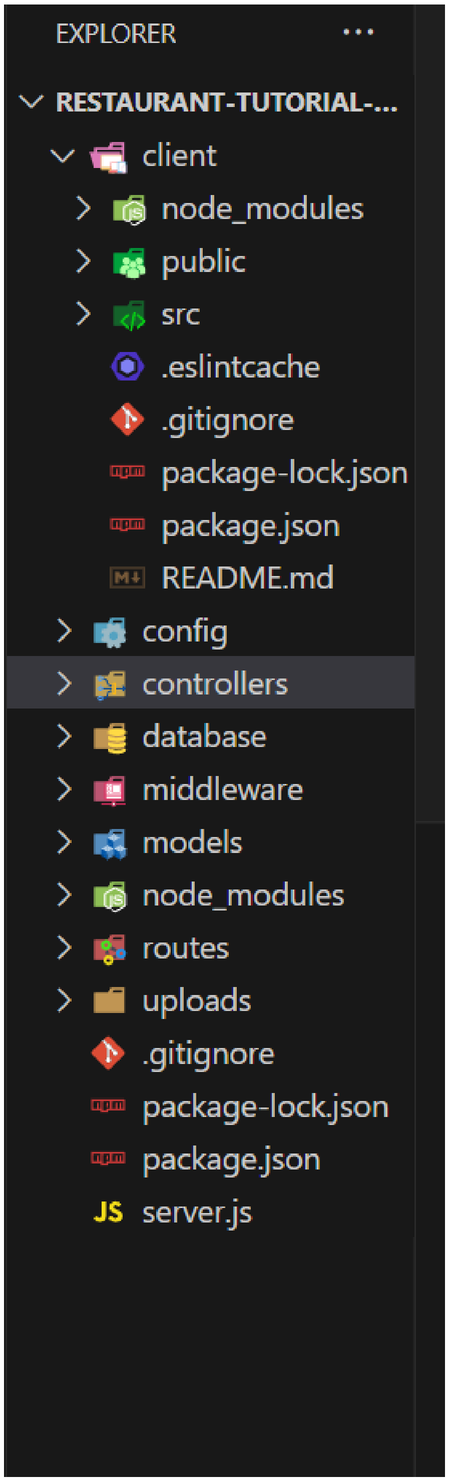
Q1 Implement best Suited Architecture for this application.

- **MVC (Model View Control):**
  I used **MVC (Model View Control)** Architecture in my **E-commerce Application** where I separated our coding into three parts modules (parts).

- **Client-Server Architecture:**
  I used **Client-Server Architecture** in my **E-commerce Application** where I used **Express JS** for Server side and **React JS** for Client side and **Mongo DB** for database so that I could save information of add-to-cart, orders, billing Payments, SignUp / SignIn and Inventory etc.

Output:

In this output, you may see client server architecture modules as well as MVC Architecture.



```
EXPLORER ...
RESTAURANT-TUTORIAL-...
  client
    node_modules
    public
    src
    .eslintcache
    .gitignore
    package-lock.json
    package.json
    README.md
  config
  controllers
  database
  middleware
  models
  node_modules
  routes
  uploads
  .gitignore
  package-lock.json
  package.json
  server.js
```

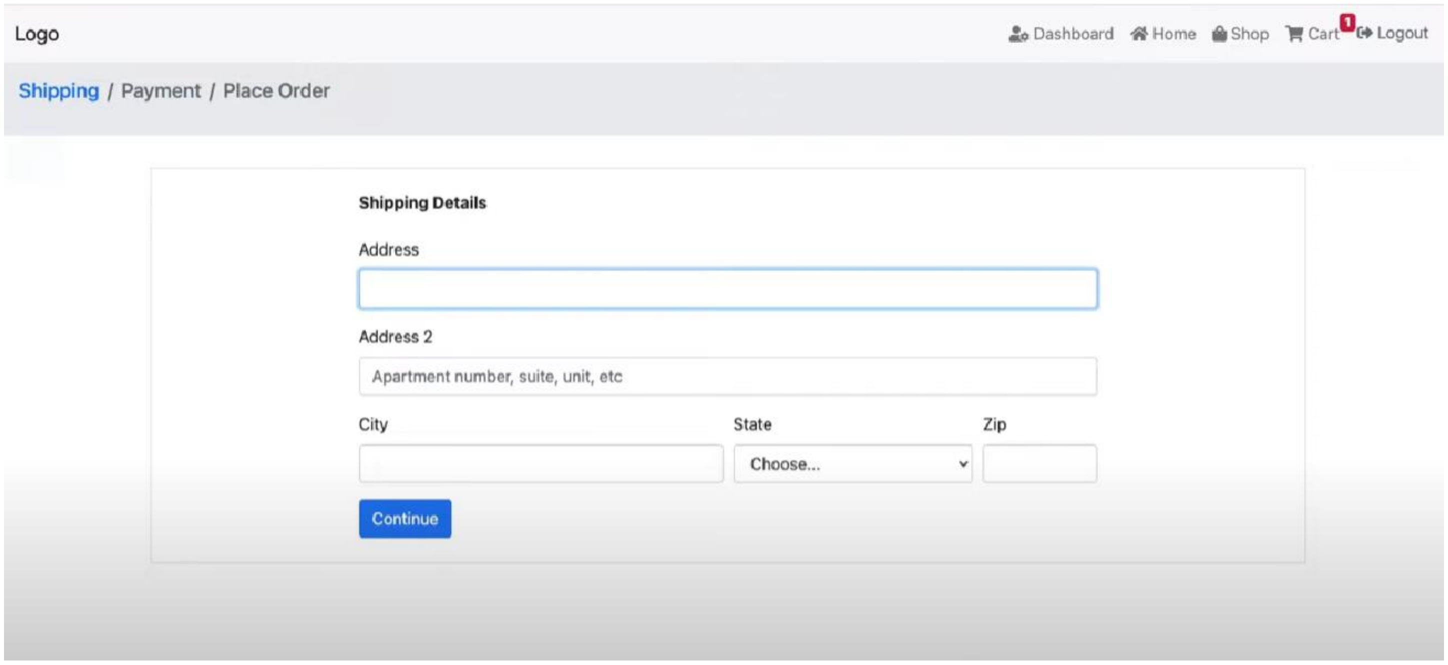
Q2 Implement best Suited Architectural Patterns and Design Patterns for this Application.

- **Shipping Processes:**

Explanation:

In this Shipping Process, We will take two addresses from Users as well as City, State and Zip Code so that We could deliver his order into his home.

Output:



Source Code:
Decorator Design Pattern:

I used **Decorator Design Pattern** For this Shipping Process, here is the code of only decorator design pattern.

```
class ShippingProcess {
    ship() {
      throw new Error('ship() method must be implemented');
    }
  }
```

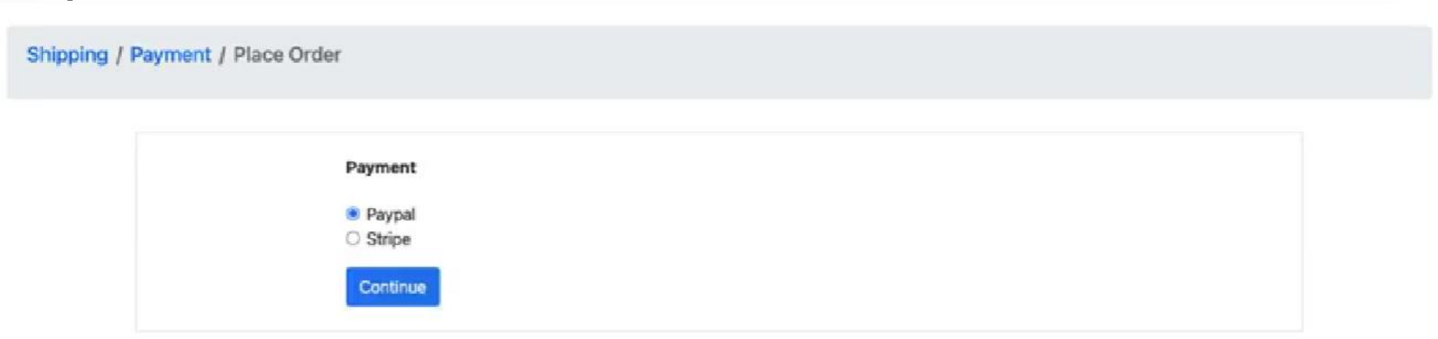
```javascript
class BasicShipping extends ShippingProcess {
  ship() {
    console.log('Shipping the product via basic shipping...');
  }
}

class TrackingDecorator extends ShippingProcess {
  constructor(shippingProcess) {
    super();
    this.shippingProcess = shippingProcess;
  }

  ship() {
    this.shippingProcess.ship();
    console.log('Adding tracking to the shipment...');
  }
}

class InsuranceDecorator extends ShippingProcess {
  constructor(shippingProcess) {
    super();
    this.shippingProcess = shippingProcess;
  }

  ship() {
    this.shippingProcess.ship();
    console.log('Adding insurance to the shipment...');
  }
}

const basicShipping = new BasicShipping();
basicShipping.ship();
```

```
  const trackedShipping = new TrackingDecorator(basicShipping);
  trackedShipping.ship();

  const insuredShipping = new InsuranceDecorator(basicShipping);
  insuredShipping.ship();

  const trackedAndInsuredShipping = new TrackingDecorator(new
InsuranceDecorator(basicShipping));
  trackedAndInsuredShipping.ship();
```

- **Payments Gateways:**

When user fill form of Shipping Process, then User come in **Payment Process** and select the payment gateways. I added Two Payment Gateways one is Paypal and second is Stripe. If one is fail the user can select another payment process.

Output:



Shipping / Payment / Place Order

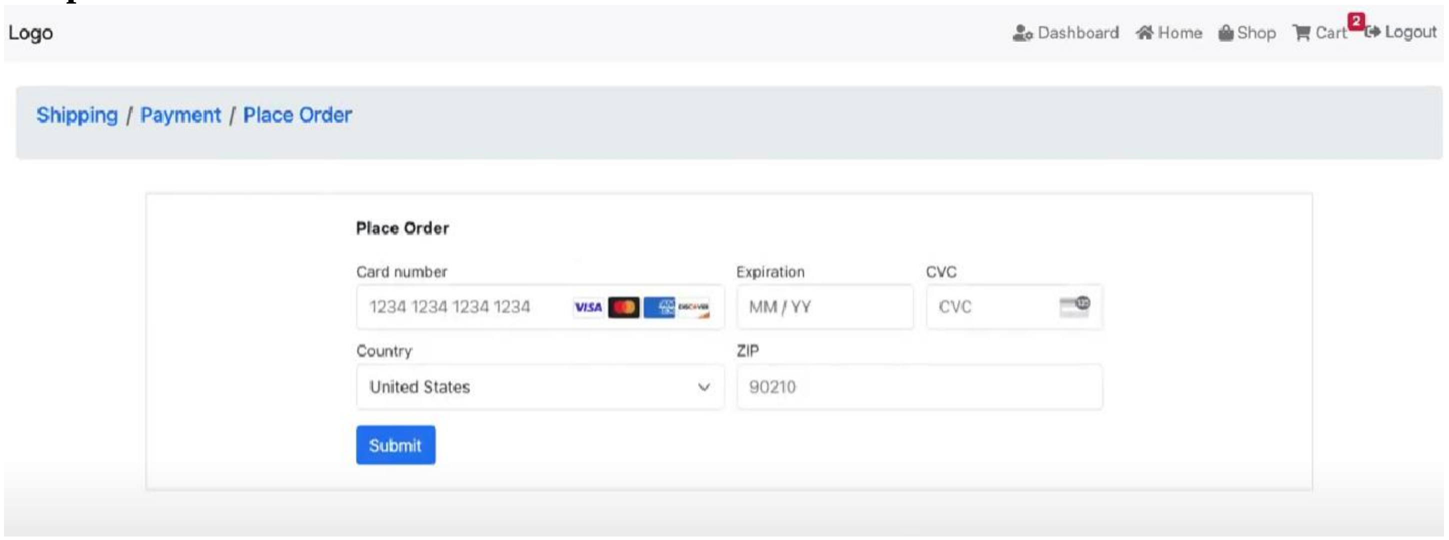
Payment

[● Paypal] ○ Stripe

Continue

When User select any Account then the information form is displayed as shown below where User give information about the **Card Number, Expiration, CVC, Country and Zip Code**.

Output:



Place Order

Card number: 1234 1234 1234 1234

Expiration: MM / YY

CVC: CVC

Country: United States

ZIP: 90210

Source Code:
Factory Pattern:

I used Factory Design Pattern for payment Process so that We could make one parent class which is PaymentGateway and make two subclasses which are child of parent class, the name is Stripe and Paypal

```
class PaymentGateway {
    processPayment(user, amount) {
      processPayment(amount) {
    throw new Error('processPayment() method must be 
implemented');
  }

    }
  }

  class PaymentGatewayFactory {
    static createPaymentGateway(type) {
      switch (type) {
        case 'stripe':
          return new StripePaymentGateway();
        case 'paypal':
          return new PaypalPaymentGateway();
        default:
          throw new Error('Invalid payment gateway type');
      }
    }
  }

  class StripePaymentGateway extends PaymentGateway {
    processPayment(amount) {
      console.log(`Processing payment of $${amount} via
Stripe...`);
    }
const { total } = req.body;

    const paymentIntent = await stripe.paymentIntents.create({
        amount: total,
        currency: 'usd',

  }

  class PaypalPaymentGateway extends PaymentGateway {
    processPayment(amount) {
      console.log(`Processing payment of $${amount} via
Stripe...`);
    }
const { total } = req.body;

    const paymentIntent = await stripe.paymentIntents.create({
        amount: total,
        currency: 'usd',
  }

  const paymentGatewayFactory = new PaymentGatewayFactory();
  const paymentGateway =
paymentGatewayFactory.createPaymentGateway('stripe');
```

```
  paymentGateway.processPayment(user, 100);
```

- **Notifications Module:**

Observer Design Pattern:

I used Observer Design Pattern for Notification Module.

```
class NotificationObserver {
    constructor() {
      this.observers = [];
    }

    addObserver(observer) {
      this.observers.push(observer);
    }

    removeObserver(observer) {
      this.observers = this.observers.filter((obs) => observer
!== obs);
    }

    notify(data) {
      this.observers.forEach((observer) =>
observer.update(data));
    }
  }

  class EmailNotification {
    update(data) {
      console.log(`Sending email notification to
${data.user.email} about ${data.type}`);
    }
  }

  class SMSNotification {
    update(data) {
      console.log(`Sending SMS notification to ${data.user.phone}
about ${data.type}`);
    }
  }

  class NotificationService {
    constructor() {
      this.notificationObserver = new NotificationObserver();
    }

    scheduleNotification(user, type) {
      const data = { user, type };
      this.notificationObserver.notify(data);
    }

    addObserver(observer) {
      this.notificationObserver.addObserver(observer);
    }

    removeObserver(observer) {
      this.notificationObserver.removeObserver(observer);
    }
  }

  const notificationService = new NotificationService();

  const emailObserver = new EmailNotification();
  const smsObserver = new SMSNotification();
  notificationService.addObserver(emailObserver);
  notificationService.addObserver(smsObserver);
  // Scheduling notifications
  const user = { email: "shoaibakhter181422.com", phone:
"03212521423" };
  notificationService.scheduleNotification(user, "newsletter");

  notificationService.removeObserver(smsObserver);
  notificationService.scheduleNotification(user, "order
confirmation");
```

- **Message Queuing:**

Decorator Design Pattern:

I used Decorator Design Pattern for Message Queuing

```
class OrderQueue {
    constructor() {
      this.orders = [];
      this.observers = [];
    }

    addObserver(observer) {
      this.observers.push(observer);
    }

    removeObserver(observer) {
      this.observers = this.observers.filter((obs) => observer
!== obs);
    }

    notifyObservers(order) {
      this.observers.forEach((observer) =>
observer.update(order));
    }
```

```javascript
    addOrder(order) {
      this.orders.push(order);
      this.notifyObservers(order);
    }

    processOrders() {
      this.orders.forEach((order) => {
        // Process order asynchronously
        setTimeout(() => {
          console.log(`Order processed: ${order}`);
        }, 2000);
      });
    }
  }


class OrderProcessor {
    constructor() {}

    update(order) {

      setTimeout(() => {
        console.log(`Order processed: ${order}`);
      }, 2000);
    }
  }


const orderQueue = new OrderQueue();
const orderProcessor = new OrderProcessor();
orderQueue.addObserver(orderProcessor);

orderQueue.addOrder("Order #1");
orderQueue.addOrder("Order #2");
orderQueue.addOrder("Order #3");
orderQueue.processOrders();
```

- **Affiliate Program Module:**

Mediator Design Pattern

I used Mediator Design Pattern for Affiliate Program.

```
class AffiliateReferral {
    constructor(affiliateId, referralId, commissionAmount) {
      this.affiliateId = affiliateId;
      this.referralId = referralId;
      this.commissionAmount = commissionAmount;
    }
  }

  class AffiliateProgram {
    constructor() {
      this.referrals = [];
      this.affiliates = [];
    }

    addReferral(referral) {
      this.referrals.push(referral);
      this.notifyAffiliates(referral);
    }

    addAffiliate(affiliate) {
      this.affiliates.push(affiliate);
    }

    getAffiliateReport(affiliateId) {
      const affiliateReferrals = this.referrals.filter(referral
=> referral.affiliateId === affiliateId);
      let totalCommission = 0;
      affiliateReferrals.forEach(referral => totalCommission +=
referral.commissionAmount);
      return `Affiliate Report: Affiliate ID ${affiliateId},
Total Commission ${totalCommission}`;
    }

    notifyAffiliates(referral) {
      const affiliateReferrals = this.referrals.filter(ref =>
ref.affiliateId === referral.affiliateId);
      affiliateReferrals.forEach(ref => {
        const affiliate = this.affiliates.find(aff =>
aff.affiliateId === ref.affiliateId)
        affiliate.receiveNotification(ref);
      });
    }
  }


  class Affiliate {
    constructor(affiliateId, name, email) {
      this.affiliateId = affiliateId;
      this.name = name;
      this.email = email;
    }

    receiveNotification(referral) {
      console.log(`Affiliate Notification: ${this.name}, Referral
ID ${referral.referralId}, Commission
${referral.commissionAmount}`);
    }
  }
  const affiliateProgram = new AffiliateProgram();
const affiliate = new Affiliate(1, "Shoaib Akhter",
"shoaibakhter.com");
affiliateProgram.addAffiliate(affiliate);

affiliateProgram.addReferral(new AffiliateReferral(1, 1001, 50));
```

- **Extensions Module:**

Strategy Design Pattern

I used Strategy Design Pattern for Extension Module because it can be used to implement different security protocols for the endpoints.

```
class Extension {
    constructor(name, endpoint, securityProtocol) {
      this.name = name;
      this.endpoint = endpoint;
      this.securityProtocol = securityProtocol;
    }

    setSecurityProtocol(securityProtocol) {
      this.securityProtocol = securityProtocol;
    }

    encryptData(data) {
      return this.securityProtocol.encrypt(data);
    }

    decryptData(data) {
      return this.securityProtocol.decrypt(data);
    }
  }
```

```javascript
  class ExtensionDecorator {
    constructor(extension) {
      this.extension = extension;
    }

    addAuthentication(authentication) {
      this.authentication = authentication;
    }

    getEndpoint() {
      return this.extension.endpoint;
    }


  }

  const extension = new Extension("API",
"<https://codingshub.com/api>", new HTTPS());
  const extensionDecorator = new ExtensionDecorator(extension);
  extensionDecorator.addAuthentication("Bearer Token");
  extensionDecorator.request("<https://codingshub.com/api/data>")
;
```