A close up of a logo

Description automatically generated

**BBEZ**

**IS213 Enterprise Solution Development**

**G6 Team3**

**Assignment**

Lim Jie Min (01363050)

Min Htet Hein

Natasya Queentina (01369694)

Shourya Goenka (01365404)

Tan Jing Da Joshua (S9535154I)

Tiffany Tan Min (01377501)

# **Introduction** Based on Grab’s data in 2018, bubble tea purchases in Singapore had increased at a rate of 3000%. The increased in demand for bubble tea may be a nightmare for some bubble tea lovers out there, as they may queue for hours just to get their hands on their favourite bubble tea. Some may even sacrifice their entire lunch break to queue for their ‘daily dose’ of bubble tea. However, long queues will no longer be a problem with BBEZ as it aims to improve customers’ bubble tea ordering experience.

# BBEZ is a website that allows customers to browse and order various drinks, like bubble tea, which are featured in the BBEZ menu. In addition, BBEZ is equipped with a feature that checks the status of the drink in real time so that customers can resume with their other activities while waiting for their drink without having to worry about missing their order collection. Moreover, a notification will be given once their drink is ready for collection so the chances of getting a melted iced-drink or a cold warmed-drink are greatly reduced.

# **Technical Overview Diagram**

|  |  |
| --- | --- |
| A screenshot of a cell phone  Description automatically generated | A screenshot of a cell phone  Description automatically generated |
| Technical Diagram 1.0 | Technical Diagram 2.0 |

# **User Scenarios**

## **Scenario 1: Customer places order**

*A screenshot of a cell phone

Description automatically generated*

*Customer places order – User Scenario Diagram*

1. OBTOS UI invokes the Menu service via HTTP GET to retrieve the menu.
2. Menu service returns the various drinks and toppings available and display in OBTOS UI.
3. Customer selects the drinks of their order by pressing buttons on OBTOS UI.
4. OBTOS UI gets the drinks and toppings information and calculates the total price.
5. Customer presses ‘Purchase’ button to make payment.
6. OBTOS UI invokes the Payment service via HTTP POST to process the order payment. It will also pass JSON data that includes customer id, e-wallet balance and order amount.
7. Payment service invokes Customer service via HTTP POST to send the updated e-wallet balance after successful payment process.
8. On successful payment, Payment service will also invoke Order service via HTTP POST to create a new order.
9. Order service returns order status to OBTOS UI via HTTP POST.
10. Order service sends Notification service about confirmed order.

### **(Micro)Services**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Service Name | Operational information (e.g., HTTP URL or AMQP exchange type and keys, if any) | Description of the functionality | Input (if any) | Output (if any) |
| Menu | [POST] /menu | Generates the menu containing drinks and toppings available |  | Returns the menu containing drinks and toppings available |
| Payment | [POST] /payment | Facilitates the payment for the order | {cid, ewallet} | {status} |
| Customer | [POST] /customer/getewallet | Sends customer ewallet information | {cid} | {ewallet} |
| Order | [POST] /order/neworder | Creates a new order in the system | {storeid, cid, status, price} | {orderid} |
| Notification | Direct Exchange with  [RKey] customer.order | Send notifications to customer | {cid, status} |  |

## 

## **Scenario 2: Cashier accepts order**

*A screenshot of a cell phone

Description automatically generated*

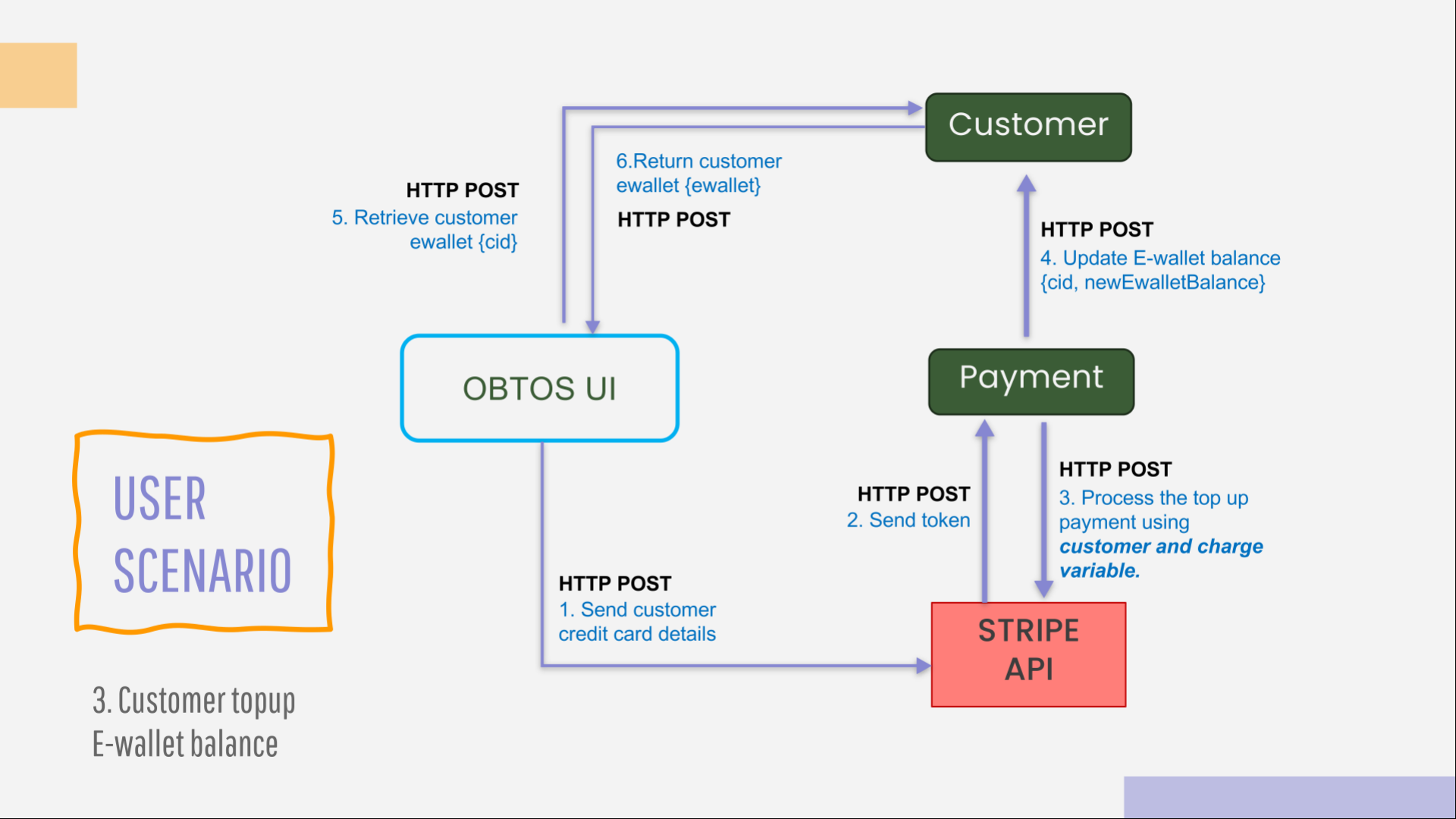
*Cashier accepts order – User Scenario Diagram*

1. Store UI retrieve all orders by order status ‘pending’ via HTTP POST.
2. Order Service will return the pending orders back to be displayed on the Store UI.
3. Cashier needs to input the ‘orderid’ in the text box on the Store UI and press ‘complete’ button.
4. The button will invoke the Order Service via HTTP POST to update the new status of the order.

### **(Micro)Services**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Service Name | Operational information (e.g., HTTP URL or AMQP exchange type and keys, if any) | Description of the functionality | Input (if any) | Output (if any) |
| Order | [POST] /order | Retrieve pending orders | {status= ‘Pending’} | {list of orders} |
| [POST] /order | Send confirmed order | {orderid} |  |

## **Scenario 3: Customer top up E-wallet with an external API (Stripe)**



*Customer top-up E-Wallet balance – User Scenario Diagram*

1. Customer will enter the top-up amount and press ‘pay’ button.
2. Customer will need to enter their credit card credentials and press ‘pay’ button.
3. The credentials entered in OBTOS UI will be send to STRIPE API for verification and tokenization.
4. Payment Service will be invoked via HTTP POST after getting the token to create *customer* and *charge variables*.
5. Payment Service invokes STRIPE API via HTTP POST to process top-up payment.
6. Payment Service invokes Customer Service to update the e-wallet balance of the user via HTTP POST.
7. OBTOS UI will invoke Customer Service for their details
8. Customer Service return the information

### **(Micro)Services**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Service Name | Operational information (e.g., HTTP URL or AMQP exchange type and keys, if any) | Description of the functionality | Input (if any) | Output (if any) |
| Payment | [POST] /topuppayment | Process top up payment | {cid, ewallet} |  |
| [POST] /customer/updateewallet | Update ewallet balance | {cid} |  |
| [POST] /payment | Return payment status | {cid, ewallet} | {status} |
| Customer | [POST] customer/getewallet | Sends customer details | {cid} | {ewallet} |
| STRIPE | [POST] /topuppayment | Process top up payment | {topUpAmt} |  |

### **Beyond the Labs**

Dealing with credit card payment can be a hassle and at times, unsecured. Therefore, in order to avoid any unwanted issues and to maintain the security of our customers’ private details, we decided to incorporate STRIPE to help us to process any credit card payments made.

STRIPE is a third-party company that provides online payment processing services for internet business. To use STRIPE service, we will need to:

1. Pip install stripe

A screenshot of a cell phone

Description automatically generated

1. In ewallet.html, we will add STRIPE checkout button using the code above.
2. A picture containing food, drawing

   Description automatically generatedThe data-key contains the publishable key that is used in the client-side code to tokenize payment information, using Stripe Checkout. It can only be used to create tokens which is a representation of a payment source without any sensitive information attached to it.   
     
     
   Therefore, upon clicking on the checkout button, a pop up will appear and customer will need to enter their email and credit card credentials. This information will be sent over to STRIPE server for validation. In return, we will receive a token for us to use in our specified location, topuppayment, to charge the payment.
3. In our payment service, we will need to assign our stripe.api\_key with our secret key. The secret key is used in our backend code to send any other request to STRIPE API.   
     
   
4. A picture containing black, white

   Description automatically generatedIn topuppayment route, we will first create a customer (*stripe.Customer.create*) that is based on customer email and token that was being created.
5. We will then create charge (*stripe.Charge.create*) to process the payment. It will need the customer id that we create in step 5, amount, currency and the description of the payment.   
     
     
     
     
     
   A screenshot of a cell phone

   Description automatically generated
6. Hence, STRIPE will run the two variables mentioned above, customer and charge. Upon successful process, we will able to add the top up amount to the customer current e-wallet balance and return the updated amount back to Customer Service.
7. Since payment.py is interacting with an External API (STRIPE) and customer.py. Therefore, payment.py is considered as a composite microservice.