THE FOSHI STONKS

IS213 Enterprise Solution Development
Section G8T6
Assignment

Team members:
Cao Wanyue (01370623)
Ching Jia Wei (01335452)
Chu Wei Hao (01360083)
Lim Chee Xiong (01324733)
Zhang Xiaoyue (01368811)

Introduction

ESD G8T6 presents to you -- Foshi Stonks -- a Game that is all about STONKS!

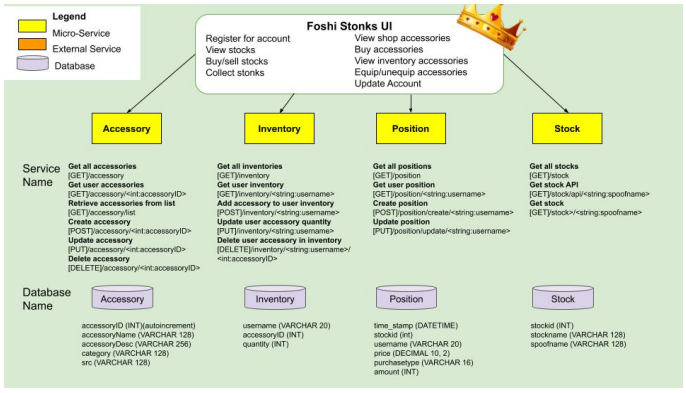
Business Scenarios:

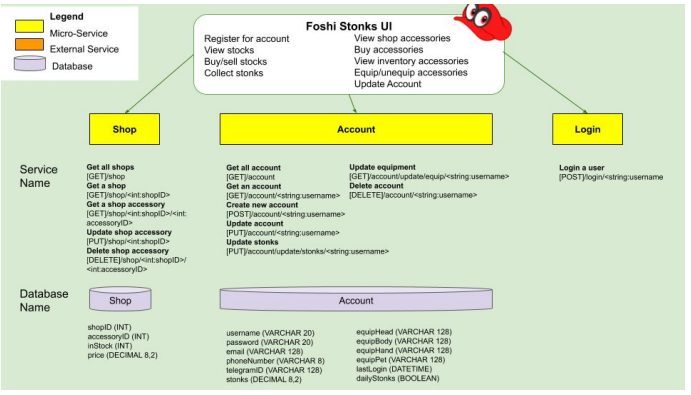
The objective of this game is to let Foshis(users) get as much stonks (in game currency) as possible to buy and wear cool accessories to flex on other Foshis (the users of this application). Users could always buy or sell stocks with stonks to gain more stonks. The stocks we provided in this game are based on the real-world stock market, which have spoof names displayed so users can't find the exact stock they are investing in. Alternatively, users can collect $10 daily stonks by clicking in the coin upon login.

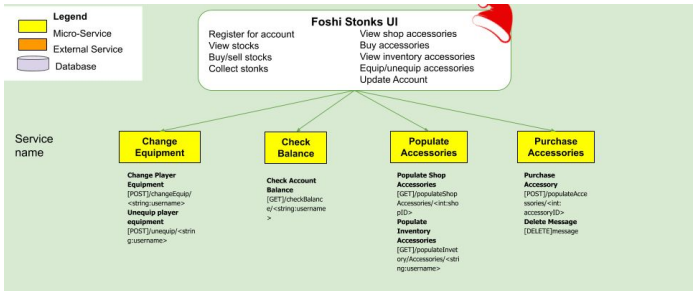
User Scenarios:

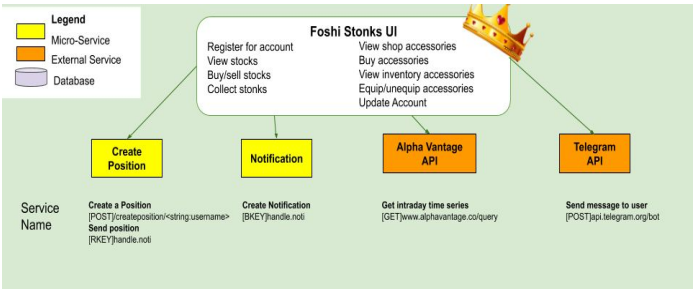
Services Explained in this report	Services Developed but not discussed in this report
1. User views stock that can be bought 2. User buys stocks 3. User views shop accessories 4. User buys accessories 5. User equips accessories	• User register for an account • User view stocks that can be sold • User sell stocks • User collects daily stocks • User views inventory accessories • User unequipping accessories • User updates account

Technical Overview Diagram



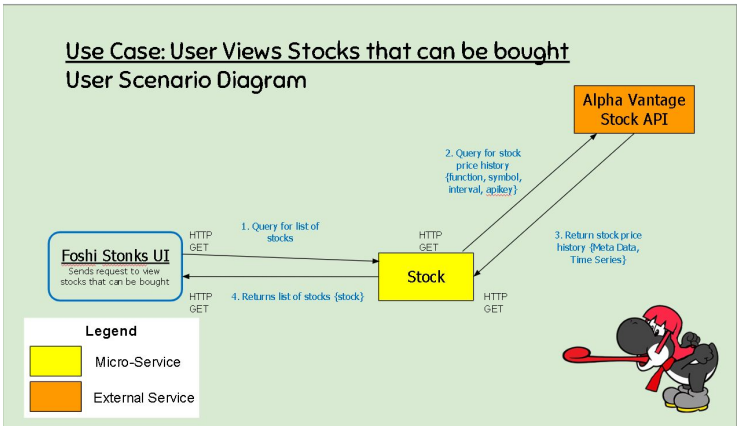






User Scenarios

[1. View stocks that can be bought]



Viewing stock that can be bought - User Scenario Diagram

1. User starts by clicking on the “Stock” page from the home.php page and lands on the stock page, then the UI will invoke the Stock service via HTTP GET to get the list of stocks that can be bought by the user.
2. Upon receiving the UI request, the Stock service invokes the external Alpha Vantage Stock API to fetch a list of stock and its stock price history
3. The Alpha Vantage Stock API returns the stock price history back to Stock service.
4. The Stock service will process the stock and stock price history data and return the list of stock with the stock price history to the UI

(Micro)Services

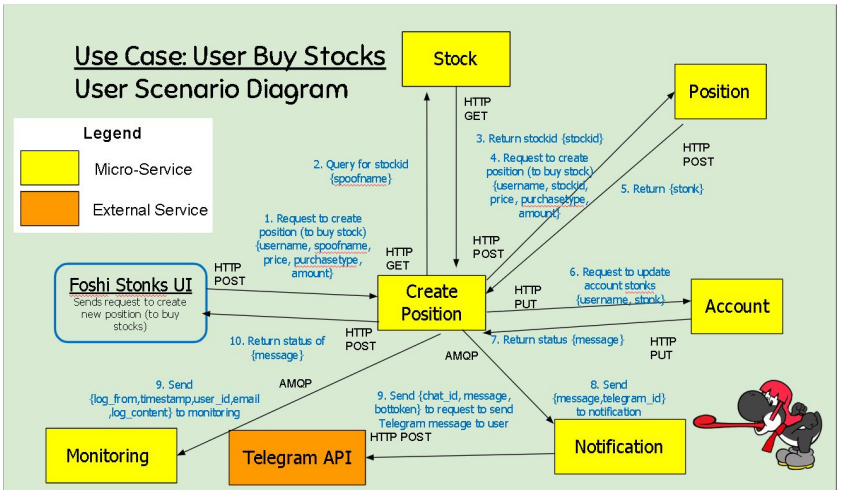
Service Name	Operational information (e.g., HTTP URL or AMQP exchange type and keys, if any)	Description of the functionality	Input (if any)	Output (if any)
Stock	[GET] /stock/api/<string:spoofname>	Populating the list of stocks available for buying and calling the Alpha Vanatage API and return the result to UI	spoofname in HTTP URL	list of stock with stock price history {stock}
Alpha Vantage API	https://www.alphavantage.co GET /query?function=TIME_SERIES_INTRADAY&symbol=<**stockname**>&interval=1min&apikey=2BAMKY2DJ4ZKBEK2	Populate a list of stock price history and time stamps of the price and return back to Stock service	function,symbol,interval, apikey	{Meta Data, Time Series}

Beyond the Labs

1. Invoke Alpha Vantage API

Alpha Vantage provides real time and historical stock data. It has numerous APIs to obtain stock data. For our project, we used TIME_SERIES_INTRADAY function to obtain stock data in 1-minute intervals. We used HTTP GET to send a request to Alpha Vantage API, passing in {function, symbol (which is the name of the stock), interval, api key}. The free API key was obtained by our group for this project (which can make 5 requests a minute). Alpha Vantage API responds with {Meta Data, Time Series}. MetaData has basic stock information, as well as the timestamp of the request, while Time Series contains an array of stock prices taken in 1 minute intervals.

[2. User Buy Stock]



User Buy Stock - User Scenario Diagram

1. User starts by clicking on the "buy" button on the UI, then the UI invokes the Create Position Composite Service via HTTP POST to create a position
2. Upon receiving the UI request, the Create Position service invoke the Stock service via HTTP GET to query for the stockid that user wants to purchase
3. The Stock service returns the stockid back to the Create Position Service.
4. Upon receiving the stockid from Stock service, Create Position service invokes the Position Service via HTTP POST to create a position record in the position database.
5. The Position service will create a new position data in create position database and calculate the total stonk price of stock that user wants to purchase and return the cost amount back to Create Position service.
6. Upon receiving the stonks amount, Create Position service will then invoke Account Service via HTTP PUT to update the user's stonk count in account.
7. The Account service will calculate the final amount left in the user's account and update the account database accordingly. It will then return a status message to Create Position service indicating if the update account is successful.
8. Upon receiving the account update status from Account Service, the Create Position will then invoke the Notification Service via AMQP direct message to send a message to the user informing them about successful stock purchase.

9. The Create Position Service will then proceed to invoke Monitoring service via AMQP direct message to store the monitoring logs inside monitoring database.At the same time, the Notification service will invoke the Telegram API via HTTP POST to send telegram message to user about successful stock purchase.
10. The Create Position service will return the status of stock purchase back to UI

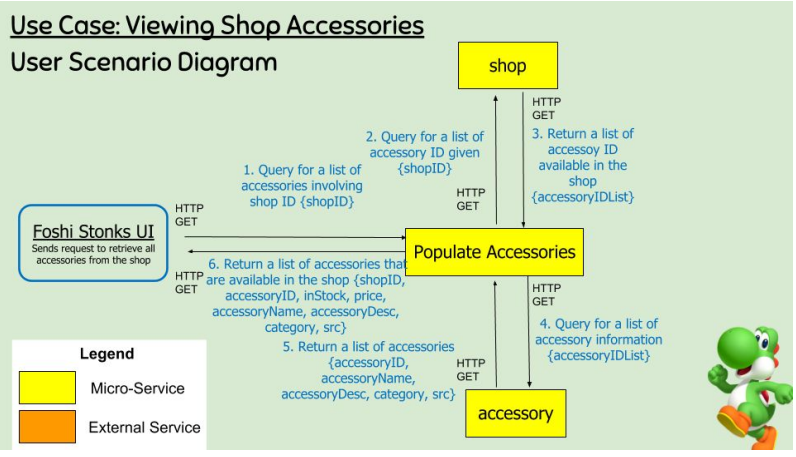
(Micro)Services

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)
Create Position	[POST] /createposition/<string:username>	Served as a composite microservice to handle stock transactions and to invoke the different services such as Stock, Position, Account, Notification and Monitoring and return the stock transaction results to the UI.	{username, soofname, price, purchasetype, amount}	transaction status {message}
Stock	[GET] /stock/<string:spoofname>	Query Stock database to obtain stock's ID that user wants to buy and return to Create Position service	spoofname in HTTP URL	{stockid}
Position	[POST] /position/create/<string:username>	Create a new position for user in database and calculate the total stonk price of stock that user wants to purchase and return the cost amount to Create Position service	{username,stockid, price,purchasetype ,amount}	{stonk}
Account	[POST] /account/update/stonks/<string:username>	Update user's stonk amount in account database and return the update status to Create Position Service	{username,stonk}	update account status {message}
Notification	[AMQP] Type: direct Keys: handle.noti	Receive message and user's telegram and invoke the Telegram API with the input received	{message,telegram_id}	-
Telegram API	https://api.telegram.org POST /bot<bottokken>/sendmessage?chat_id=<chatid>&text=<message>	Send telegram message to user notifying them about successful stock transaction	{chat_id,message, bottoken}	-
Monitoring	[AMQP] Type: direct Keys: handle.monitoring	Store monitoring log in monitoring database to keep a record of what is happening	{log_from,timestamp,user_id,email,log_content}	-

Beyond the Labs

1. Invoke Telegram API
2. Create Position composite service is created to improve the communication between microservices

[3. View Shop Accessories]



Viewing shop accessories that can be bought - User Scenario Diagram

A user try to view the accessories that he/she can buy from the shop:

1. The user clicks on the Shop.php page to invoke the Populate Accessories composite service, via HTTP GET, to get a list of accessories in the shop.
2. The Populate Accessories composite service invokes the Shop service, via HTTP GET, to get a list of accessory IDs in that shop from the Shop db.
3. The Shop service returns a list of accessory ID available in the shop to the Populate Accessories composite service.
4. Upon receiving the list of accessory ID, the Populate Accessories composite service invokes the Accessory service, via HTTP GET, to get all accessory details .
5. The Accessory service receives the list of the accessory ID and returns a list of accessory details from the Accessory db.

6. The Populate Accessories composite service receives a list of accessory details from the shop then returns the accessories details back to the UI.

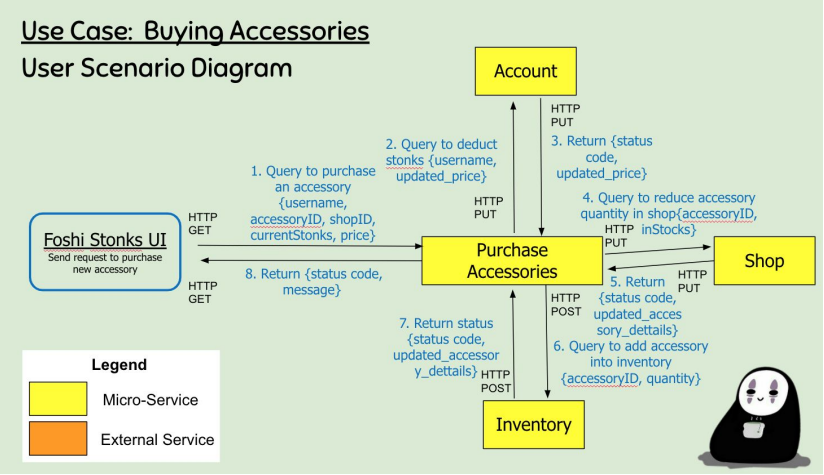
(Micro)Services

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)
Populate Accessories	[GET] /populateShopAccessories/<int:shopID>	Populate shop accessories invokes Shop and Accessory services to retrieve accessories and returns them to UI	{shopID}	{list of accessories}
Shop	[GET] /Shop/<int:shopID>	Get a list of the accessory IDs from the particular shop	{shopID}	{accessoryIDList}
Accessory	[GET] /accessory/list	Get a list of accessory details	{accessoryIDList}	{list of accessories}

Beyond the Labs

1. Populate Accessories composite service is created to improve the communication between Shop and Accessory microservices

[4. Buy accessories from the shop]



Buying accessories from the shop - User Scenario Diagram

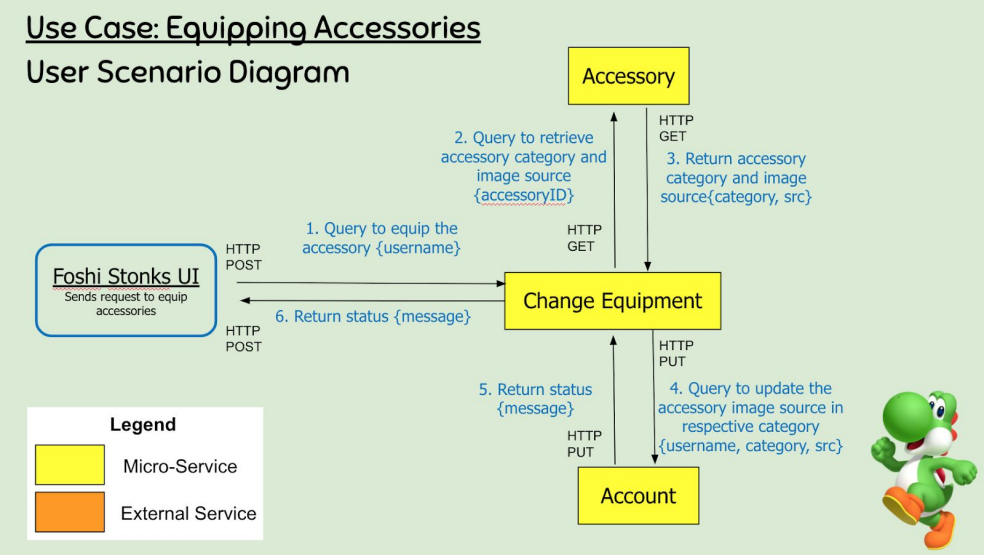
1. After viewing the accessories in the shop, the user decides to use the UI to purchase an accessory. Then the UI invokes the Purchase Accessories service via HTTP GET to buy the accessory.
2. Upon receiving a UI request, the Purchase Accessories receives necessary accessory information and then it invokes the Account service via HTTP PUT to deduct Stonks from the user's account
3. The Account service calculates the remaining Stonks via the Account db and returns a status code and the updated Stonks to the Purchase Accessory service.
4. The Purchase Accessories service invokes the Shop service via HTTP PUT to reduce the accessory quantity in the shop.
5. The Shop service deducts the quantity from the Shop db and returns a status code and the updated quantity to the Purchase Accessory service.
6. Then the Purchase Accessories service invokes the Inventory service via HTTP POST to add the accessory into the user's inventory.
7. The Inventory service adds the accessory to the Inventory db and returns a status code and the accessory details to the Purchase Accessory service.
8. Upon receiving all the updated accessory details from the Shop, Inventory and Account services, the Purchase Accessories service returns a successful/error message and accessory details back to the UI.

(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)
Purchase Accessories	[POST] /purchaseAccessories/<int:accessoryID>	Purchase shop accessories invokes Shop, Inventory and Account services to purchase the accessory and returns the message to UI	{username, accessoryID, shopID, currentStonks, price}	Return message (message: successful / error), 200/500
Shop	[PUT] /Shop/<int:shopID>	Update the stock (the quantity of the accessory in the shop) ** Assumption: user allows to buy 1 accessory at a time.	{shopID, accessoryID, inStock}	{the updated accessory from the shop } and {status code}

Account	[PUT] /account/ <string:username>	Deduct the price of the accessory and update the user's stonks	{username, updated_stonks}	{the updated stonks of the user} and {status code}
Inventory	[POST]/inventory/<string:username>	Add new accessory that bought by user into the user's inventory	{username, accessoryID, quantity}	{list of accessories in inventory, status code}

[5. Equip Accessories]



Equip avatar with accessories- User Scenario Diagram

1. After viewing the accessories owned, the user clicks an accessory on UI to equip that accessory. Then the UI invokes the Change Equipment composite service via HTTP POST to request.
2. Upon receiving a UI request, the Change Equipment retrieves information from Accessory atomic service via HTTP Get. So UI could display with correct image source and category position
3. Meanwhile, the Change Equipment also updates this user's binded equipment image source by connecting with Account atomic service via HTTP PUT

(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)**
Change Equip	[POST]/changeEquip/<string:username>	Change Equip service invokes Accessory and Account services to get image source and updates account's info to change avatar's appearance on UI	{username, accessoryID}	Return message (message: successful / error), 200/400
Accessory	[GET]/accessory/<int:accessoryID>	Retrieve the accessory image source	{accessoryID}	{all accessory info } and {status code}
Account	[PUT]/account/<string:username>	Update user's displayed equip info	{username}	Return message (message: successful / error), 200/400

Remaining Beyond the Labs not covered above

1. We have a total of 14 microservices, in particular 8 atomic microservices and 6 composite microservices.
2. Implemented 6 Composite Microservices which orchestrate atomic microservices.
3. Invoke external APIs: Telegram API and Alpha Vantage API.
4. Deploy separate Docker containers for 14 microservices that can interact with each other.
5. Utilized docker compose to automate the deployment of multiple docker containers in a single host.