Unit/Output Testing

Microservices Flask file

Should always return an HTTP message containing the response code along with a json object for some methods. This code will be primarily based on non-local SQL commands with query_database containing most of the functionality in the sense that it runs SQL commands.

Methods:

*check_list (generic helper method)

test 1

input: an array of objects and an object that is in the array

behaviour: loops through the list and compares each item to check whether or not the inputted object is in the list

output: returns the object from the list

test 2

input: an array of objects and an object that is not in the array

behaviour: since the object is not in the array, the return value does not get set

output: returns 'None'

*home_test

input: HTTP GET request with an address of http://localhost:[port]/

behavior: fetches a list of available microservices and whether they are expected to be GET POST or PUT requests

output: 200 & json object containing available microservices and type of request associated with them

*create_database

input: HTTP GET request with an address of http://localhost:[port]/database/methods/create_db

behavior: runs premade SQL commands to create the item entry, order, and receipt database

output: 200

*query_database

test 1

input: HTTP POST, http://localhost:[port]/database/methods/query_database/<valid SQL code>, json object containing optional information to include in the request

behavior: runs the given SQL code on the database

output: 200

test 2

input: HTTP POST, http://localhost:[port]/database/methods/query_database/<invalid or nonapplicable SQL code>, json object containing optional information to include in the request

behavior: fails to run the given SQL code on the database

output: 400

test 3
input: HTTP POST, same address as *1*, json object containing invalid or nonapplicable information in the request
behavior: the SQL code either fails to run or runs and just ignores unused inputs
output: 400

*query_sql (query_database helper method)

test 1
input: SQL command as a string and optionally, what database to run the command on
behavior: connects to the database file and attempts to run the SQL command on it
output: 200

test 2
input: same as before but the given database DNE
behaviour: fails to open the database
output: 400

test 3
input: valid database, invalid SQL command
behaviour: fails to run the SQL command
output: 400

Customer UI

*load_menu

test 1
input: should have no real input. this will be the base loading method for the customer's menu
behaviour: loads a menu where you can select a category to browse
output: again, no real output, but the menu being displayed as intended is indicative that it worked

test 2
input: n/a
behaviour: database error
output: display error message

*load_category

test 1
input: a valid category that contains a list of item entry objects that it should contain
behaviour: gui goes from the overall category menu to a display of the items inside that category. on the backend, the database will be queried for all of the item entries in the desired category and then they will be processed into item entry objects on the program end and fed into the GUI.
output: a list of item entry objects made up of every item entry in the inputted category

test 2

input: an invalid category that does not contain any item entries
behaviour: this should never happen since only existing categories should be displayed but it can theoretically be tested on the code end.
output: some sort of error message
test 3
input: a valid category but the database throws an error
behaviour: database throws an error
output: display error message

*load_item
test 1
input: an item entry
behaviour: should provide the name, price, picture, and addons of the item for the user. This is all stored in the item entry objects except for the picture which would need to be converted from its string format in the item entry object into a visual display.
output: none(?)
test 2
input: an invalid item entry
behaviour: a check for one of the item entry's fields fails
output: throw an error

*add_to_order (method in Order class)
test 1
Input: an item_entry object
Behaviour: should add an item with its options to an in-progress order.
output: updated order object(?)
test 2
Input: invalid data in the item entry object
behaviour: a check for one of the item entry's fields fails
Output: throw an error

*compile_order
test 1
input: an order object
behaviour: categorizes the order and prepares it as an object to be sent to the cook. specifically, this is where payment processing would come into play. refer to that section for specific methods used
output: an updated order object, including a subtotal, tax total, and final total
test 2
input: an order with bad or malformed information
behavior: fails to compile due to bad information
output: throw an error

*send_order
test 1
input: an order to be sent
behaviour: calls on several helper methods to send the user's order to the cook end and adds the user's order to the order database

output: 200
test 2
input: an order with bad or malformed information
behaviour: fails to query the database due to the order’s malformation
output: 400
test 3
input: an order to be sent but the cook end is offline
behaviour: throws a connection error to the customer end, removes the order from the database, and refunds the customer
output: 503

*access_database
test 1
input: an order to be added to the database
behaviour: queries the order database (address is not finalized) to add the order and then extracts the primary ID of the order for ‘ID association’
output: 200, json object containing the order’s primary key
test 2
input: an order to be added, but the database cannot be accessed
behaviour: database is down or something
output: 503

*encrypt_order
imports from Order Encryption (Signatures), refer to that section for actual tests

*transmit_order
test 1
input: final encrypted order to be sent
behaviour: transmits encrypted order to the cook end
output: a success if an acknowledgement is received
test 2
input: final encrypted order to be sent but cook end is down
behaviour: fails to transmit encrypted order
output: 503

Cook ui

- void addOrder(Order)
    - *test 1*
        - Input: valid Order object, containing at least one item
        - Behavior: adds order to the list of current orders, displays new list
    - *test 2*
        - Input: invalid Order object containing no items
        - Behavior: Error: empty order. Does not update list
    -
- void delOrder(idx)
    - *test 1*
        - Input: idx < Orders.size() && idx >= 0

        - Behavior: order at idx is removed from the list, display is updated to reflect this
    - *test 2*
        - Input idx >= Orders.size() || idx < 0
        - Behavior: Error: idx out of bounds. List and display are not updated

Payment Processing

*compile_payment

test 1

input: an Order object containing various item entry objects with addons and taxable as true

behaviour: calculates the subtotal, tax, and final total and stores them in the Order object

output: returns the Order object with its subtotal, tax, and final total variables filled

test 2

input: an Order object containing various item entry objects with addons and no taxable items

behaviour: same as 1

output: returns the Order object with its subtotal, a tax value of 0, and a final total equivalent to the subtotal

test 3

input: an Order object containing no item entries

behaviour: short circuits to the end and returns 0s

output: returns the Order object with a subtotal tax and final total of 0.

*refund_payment

test 1

input: an order object that has been determined to need to be refunded

behaviour: does some payment api stuff to refund the order

output: none

Order Encryption (Signatures)

- String signOrder(Order order, int privKey)
    - *test 1*
        - Input: a valid order object with at least one item
        - Behavior: calculates a signature based on both the key and object
        - Output: the signature to this object+key pair
    - *test 2*
        - Input: an invalid order with no items
        - Behavior: short circuits and returns empty string, prints error
        - Output: prints Error: empty order, returns an empty string
    -
- bool verifyOrder(Order order, String signature, int pubKey)
    - *test 1*

        - Input: a valid order, a signature generated with that order, and the public key paired with the key used for the signature
        - Output: True
    - *test 2*
        - Input: a valid order, a signature not generated with that order, and the public key paired with the key used for the signature
        - Output: False
    - *test 3*
        - Input: a valid order, a signature generated with that order, and a public key not paired with the key used for the signature
        - Output: False
    - *test 4*
        - Input: a valid order, a signature not generated with that order, and a public key not paired with the key used for the signature
        - Output: False
    -
-

User Acceptance

Customer

Use cases:

Start Program: Tap/click to wake up device (user test 1.1)
- The list of categories are displayed

Select Category: User taps/clicks on a category (user test 1.2)
- The list of sub-categories or items within the selected category are displayed

Select Item: User taps/clicks on an item (user test 1.3)
- The list of options for that item are displayed, with costs if relevant
- User can then add customization options

Add Item: User confirms adding an item (user test 1.4)
- User is returned to the main categories screen and a pop up appears to the side to confirm the item was added, containing the item's name

Checkout: User clicks proceed to checkout (user test 1.5)
- The list of items and their customization options are displayed, with costs for each item, as well as the total cost of all items in cart. Buttons for payment options are displayed

Select Payment: User selects a payment option (user test 1.6)
- Screen prompts user to swipe their card.

Provide Payment: User swipes a payment card (user test 1.7)
- If the information on the card matches the selected payment type, payment is processed, and a confirmation screen is displayed with the user's order id number. In practice, a receipt will also be printed. Screen returns to idle state. If invalid details are provided, user is returned to the checkout screen and an error

pop-up is displayed, prompting the user to try again (return the screen in user test 1.5).

Cook

- Receive Order (user test 2.1)
    - When a customer successfully Provides Payment (user test 1.7), their order appears on the cook screen, including their Order ID Number, and a list of all items in the order including any customization options, but not any relevant prices.
    - If an order is received while another order is already present on the screen, as many orders as possible are displayed at once. See User Test 2.3 for more details.
- Fulfill Order (user test 2.2)
    - User clicks on an order to highlight it, then clicks on it again.
    - The selected order is removed from the screen, all other orders remain
- Scroll (user test 2.3)
    - If more orders are active than can fit on the screen, a scroll bar appears on the side of the screen. The user can then use this scroll bar to view orders before or after those currently displayed.

Ursinus Administrator

- Fetch Daily Order Summaries (user test 3.1)
    - Whoever oversees the finances or whatever of Wismer will be able to access a daily summary of the orders taken that day