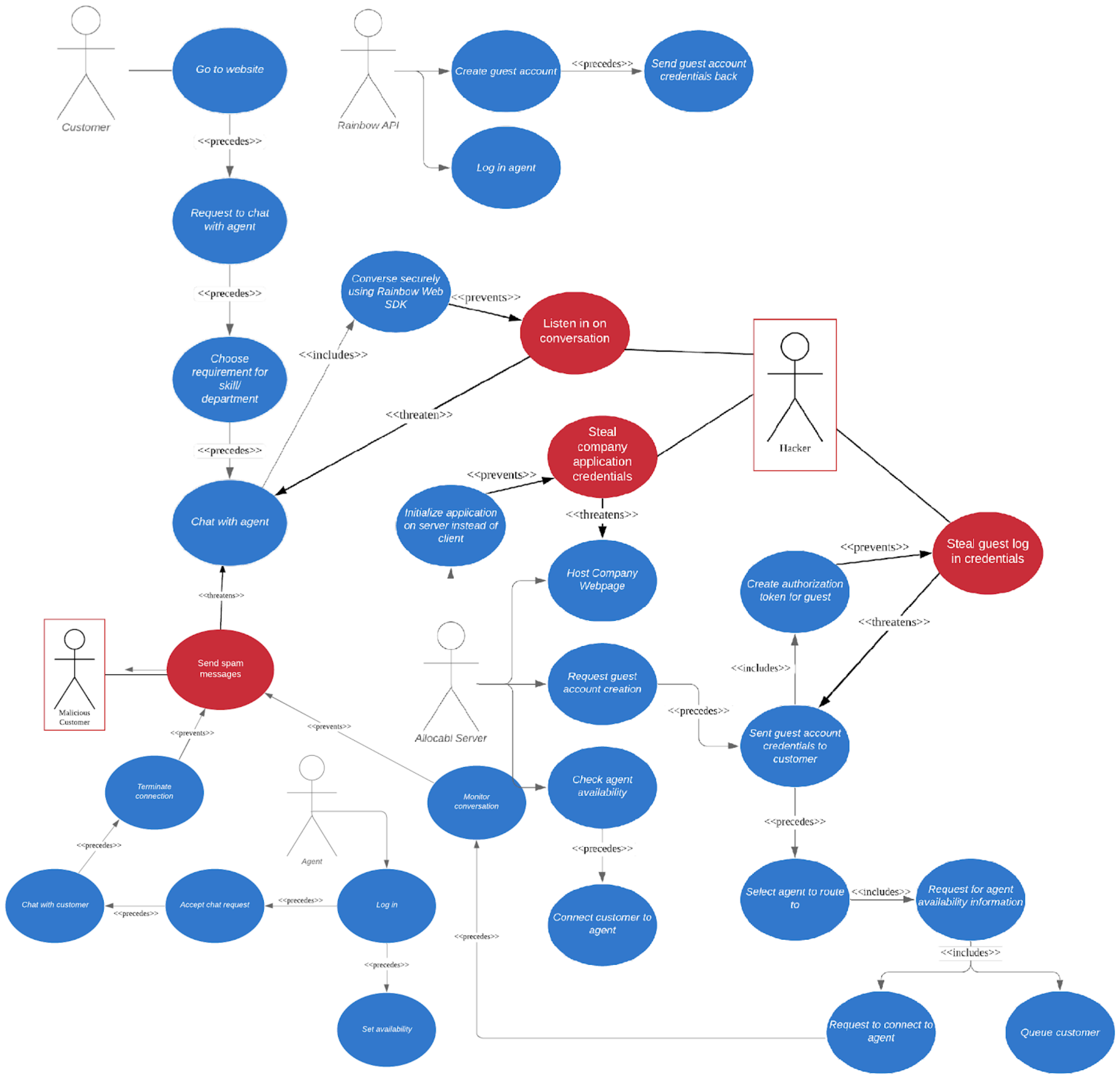
Allocabl
C1G8
Final Report

Done By:

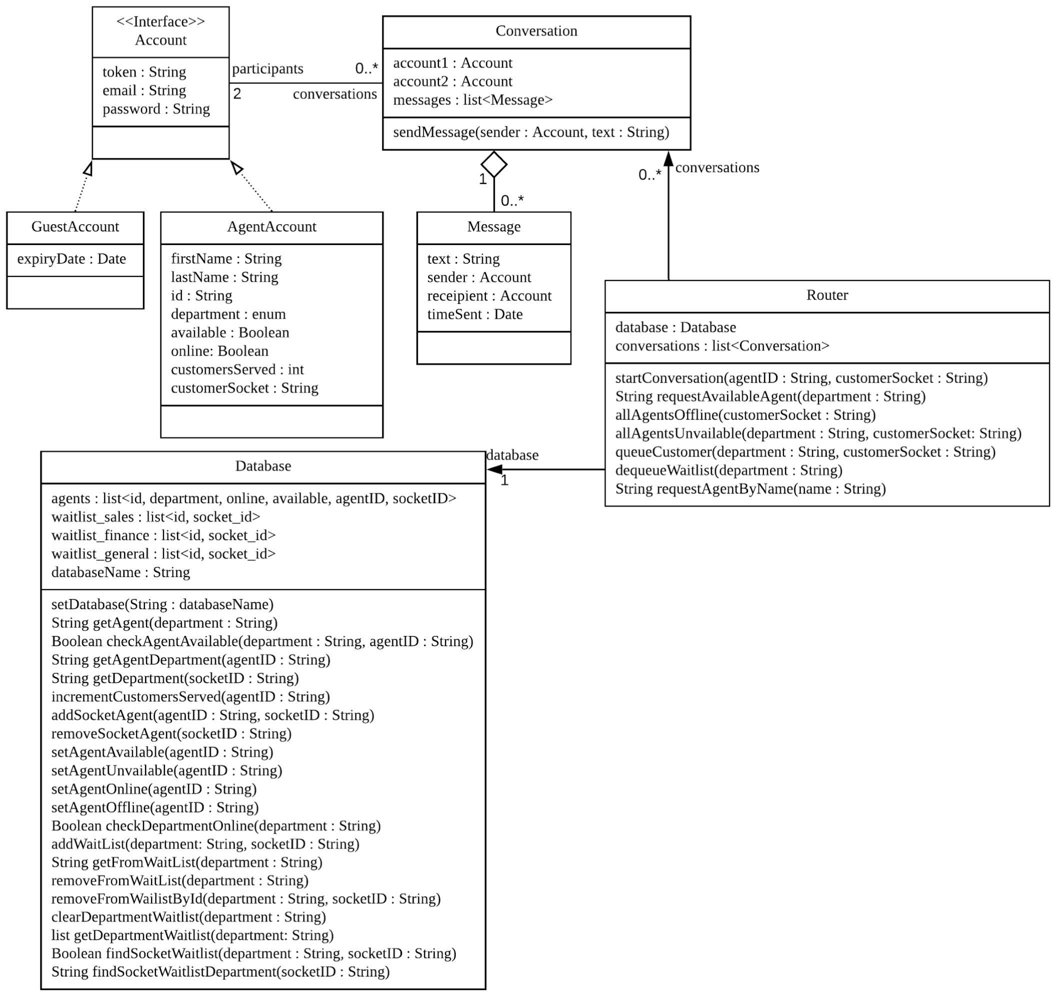
Leong EnYi	1003463
Teo Siang Chuan Daniel	1003571
Varsha Venkatesh	1003646
Koh Ling Tian	1003785

Use Case Diagram



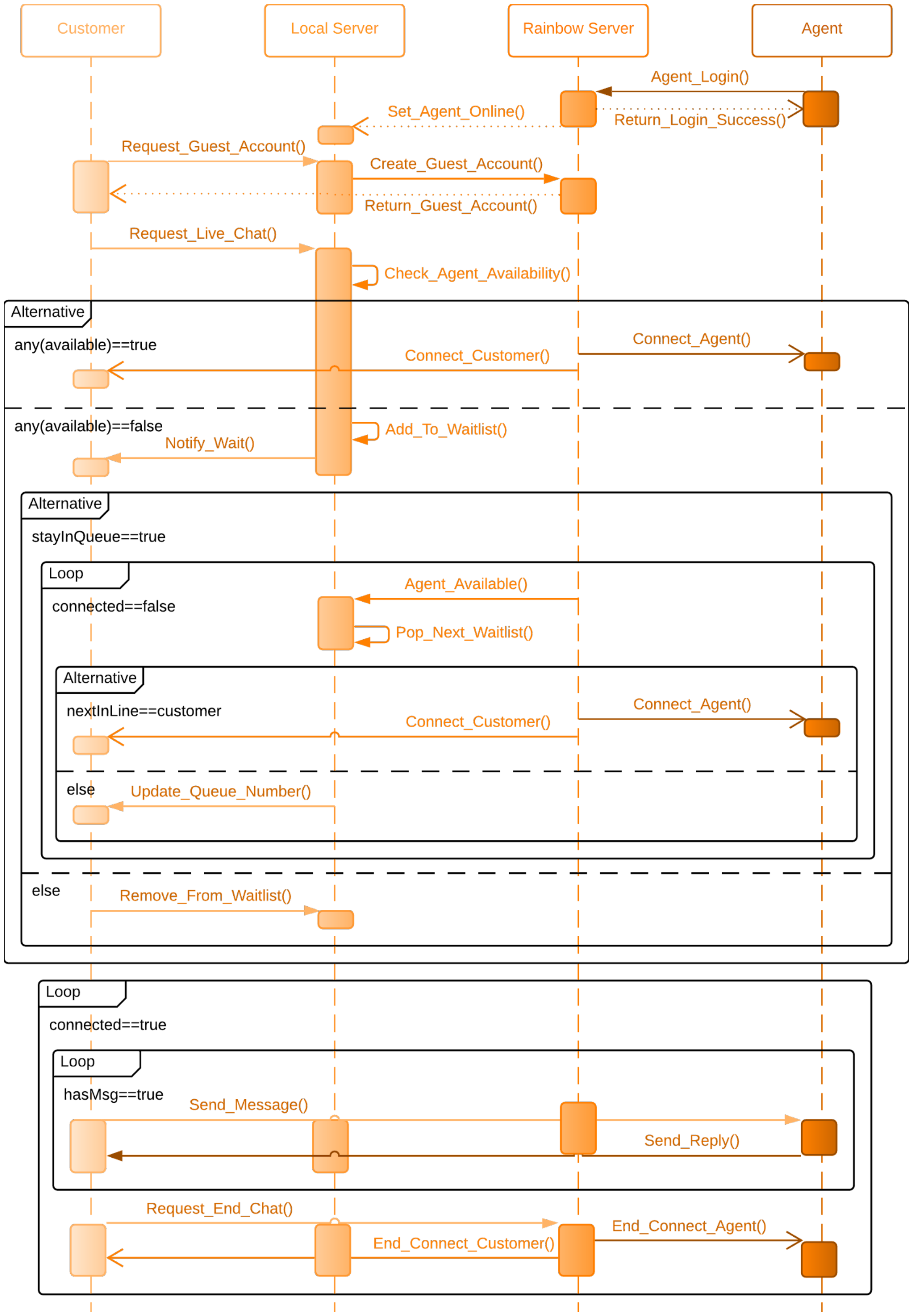
UML Diagrams

Class Diagram



<<Interface>>
Account
token : String
email : String
password : String
participants
2
0..*
conversations
Conversation
account1 : Account
account2 : Account
messages : list<Message>
sendMessage(sender : Account, text : String)
GuestAccount
expiryDate : Date
AgentAccount
firstName : String
lastName : String
id : String
department : enum
available : Boolean
online: Boolean
customersServed : int
customerSocket : String
1
0..*
Message
text : String
sender : Account
receipient : Account
timeSent : Date
0..*
conversations
Router
database : Database
conversations : list<Conversation>
startConversation(agentID : String, customerSocket : String)
String requestAvailableAgent(department : String)
allAgentsOffline(customerSocket : String)
allAgentsUnvailable(department : String, customerSocket: String)
queueCustomer(department : String, customerSocket : String)
dequeueWaitlist(department : String)
String requestAgentByName(name : String)
database
1
Database
agents : list<id, department, online, available, agentID, socketID>
waitlist_sales : list<id, socket_id>
waitlist_finance : list<id, socket_id>
waitlist_general : list<id, socket_id>
databaseName : String
setDatabase(String : databaseName)
String getAgent(department : String)
Boolean checkAgentAvailable(department : String, agentID : String)
String getAgentDepartment(agentID : String)
String getDepartment(socketID : String)
incrementCustomersServed(agentID : String)
addSocketAgent(agentID : String, socketID : String)
removeSocketAgent(socketID : String)
setAgentAvailable(agentID : String)
setAgentUnvailable(agentID : String)
setAgentOnline(agentID : String)
setAgentOffline(agentID : String)
Boolean checkDepartmentOnline(department : String)
addWaitList(department: String, socketID : String)
String getFromWaitList(department : String)
removeFromWaitList(department : String)
removeFromWailistById(department : String, socketID : String)
clearDepartmentWaitlist(department : String)
list getDepartmentWaitlist(department: String)
Boolean findSocketWaitlist(department : String, socketID : String)
String findSocketWaitlistDepartment(socketID : String)

Sequence Diagram



Testing

Unit testing

Purpose: To test individual functions and ensure that they are working as intended.
Tools: Mocha (testing framework), Chai (assertion library)
Final results: See Appendix A
Our unit tests tested the smallest units of functionality in our app. These were the database functions, which served as an API for us to communicate with our SQL database from our Node.js server, and some utility functions that we wrote. In our unit tests, each function was tested with a variety of inputs and the result was compared with the expected result. We also included invalid inputs (such as incorrect data types) and checked that the correct error was thrown.

In order to test the database functions, some setup was required, which included creating a test database before any tests were conducted. Dummy data was then added to the test database before each test, and subsequently deleted afterwards to prepare for the next test. If required, database queries were made to check the dummy data and determine whether the function under test affected the database in the desired way. This setup was performed with the help of the Mocha testing framework, which allowed us to write hooks that would be called at various steps of the testing process (before each test for example). For database functions on top of varying the inputs to the function itself, we also varied the dummy data to check for edge cases such as empty tables. A special consideration was also made to test for potential SQL attacks, even though these database functions were not called with inputs taken directly from users. This was done by analysing the query string in each specific function and providing an input that if injected directly would delete all data in the test database (a DROP query) , and subsequently ensuring that no data was affected.

Integration testing

Purpose: To test functions that involve multiple systems
Tools: Mocha (testing framework), Chai (assertion library), Sinon (test spies, mocks and stubs)
Final results: See Appendix B
For our integration tests, we tested functions that interacted with multiple other systems. These functions mainly consisted of the handler functions for events received by our server either from the Rainbow API or from the connected browsers.

The systems involved in these functions included our database, the Rainbow API, and the Socket.IO API. To test the resulting effects on the database, a similar method to the database unit tests was used where a test database was created and populated with dummy variables. To simulate calls made to the Rainbow and Socket.IO APIs, we made use of the Sinon library to replace these methods and objects with fakes. We then checked whether these fakes were called the correct number of times and with the desired parameters. Dummy database values were selected to simulate the different conditions under which these functions might be called.

Our integration tests allowed us to ensure that any code refactoring was properly handled by any dependent functions. For example, the method signature of the loginGuest function was changed to add in the functionality of being able to contact a specific agent by name. We then used our integration tests to ensure that other methods calling loginGuest were appropriately updated such that desired behavior was preserved.

System testing

Tool: Selenium for JavaScript

Test flows:

- Flow 1 : Test connection to each department
    1. Connect to website
    2. Request connection to agent (button)
    3. Select department
    4. Select 'no' for specific agent
    5. Agent connected
    6. Send message to agent
    7. End connection
    8. Repeat for each department

- Flow 2 : Test waitlist and availability
    1. Open 3 browsers (3 Clients)
    2. Request connection to agent (button)
    3. Select department (Sales)
    4. Select 'no' for specific agent
    5. 1nd Client: "You are now connected!"
       2nd Client: No agent available, put on waitlist (Queue: 2)
       3rd Client: No agent available, put on waitlist (Queue: 1)
    6. 2nd Client: Close
       1st Client: Close
       3rd Client: Connecting to Department
    7. 3rd Client: Connected
    8. End connection

- Flow 3: Test manual connect to specific agent that exists
    1. Connect to website
    2. Request connect to agent (button)
    3. Select department (Sales)
    4. Select ‘yes’ for specific agent
    5. Enter ‘sales1’ for agent name
    6. Agent exist and is available
    7. Agent connected
    8. End connection

- Flow 4: Test manual connect to specific agent that does not exist
    1. Connect to website
    2. Request connect to agent (button)
    3. Select department (Sales)
    4. Select ‘yes’ for specific agent
    5. Enter ‘financeA’ for agent name
    6. Agent does not exist

System Tests		
Test	**Behavior**	**Result**
testFlow() (Selenium)	Returns true when output for each department is correct Sales == “You are now connected!” Finance == “You are now connected!” General == “No agent is online!”	Passed
testFlow2() (Selenium)	Returns true when the waitlist and connection procedure is correct General == “You are now connected!”	Passed
testManualExist() (Selenium)	Returns true when specified agent exists and is available	Passed
testManualNotExist() (Selenium)	Returns true when specified agent does not exist	Passed

Robustness testing

Tool: Selenium for JavaScript

Test flows:

- Spam Test
    1. Connect to Website
    2. Request connection to agent (button)
    3. Select department
    4. Select 'no' for specific agent
    5. Agent connected
    6. Spam Agent with Message
    7. Client Disconnected

- Flood Test

In our flood test, we created a HTML page that used a for-loop to open 100 concurrent connections to 2 different departments, each connection asking to be connected to an agent. We then checked that each connection received an appropriate response from the server, based on the number of agents we had that were available at the moment. For example, if there were 5 available agents then 5 connections would have received a guest login token while the other 95 would be placed into the waitlist for that department. The purpose of this test was to test the server under load, as well as the concurrency behavior of our code.

Robustness Tests		
Test	**Behavior**	**Result**
testRobust() (Selenium)	Returns true when client is disconnected for spam Response == "Disconnected"	Passed
Flood Test (1 sales agent and 5 finance agents online)	1 sales customer received guest login credentials and 99 sales customers put into waitlist. 5 finance customers received guest login credentials, and 95 finance customers put into waitlist.	Passed

Implementation Challenges

Engineering Challenges

The first implementation of the communication between browsers and our server was done using HTTP requests. However upon addition of the waitlist functionality we realised that we needed a way for the server to initiate communication with the browser to update customers when an agent is available. Since HTTP requests are client driven, we migrated to using web sockets instead through the use of the Socket.IO API, which allowed bidirectional communication.

We ran into an issue where our custom modules were not being loaded because there was a circular dependency. To solve this, we shifted functionality out of some of the modules and into the main app file to stop such interdependence. For example, we used to attach the handler for a Rainbow API event in the module where we initialized the RainbowSDK. However this meant that the initialization module had to import from the handler module, which in turn imported the SDK from the initialization module. By having the main file import these two modules instead, this problem was averted.

Initially our database functions were found to be susceptible to SQL injection through our unit tests. This was fixed by ensuring that function parameter strings were properly escaped before adding them to the query. For parameters that could not be properly escaped (such as table names) we added checks to ensure that they were alphanumeric. If not, an error would be thrown.

Our flood test initially revealed that all 100 customers were being routed to the same agent, and none were being put into the waitlist. We found out that this was because our database was not being updated fast enough to reflect that an agent had already been routed. As such, subsequent requests would see the agent as still being available. In order to solve this issue we locked all critical functions with the same lock such that only one request would be processed at a time. This ensured that the database would always relay the most updated information to all incoming requests.

We had initially planned to host our app from a Google Cloud Compute Engine. However upon conducting our flood test we realised that the website became unresponsive and began to prematurely close connections, until only about 50-60 connections remained open. We attributed this to the limited resources provided by the Google Cloud Free Tier, which was the tier that our compute engine belonged to. In the end the app was hosted off of one of our own computers for the demonstration.

On the front end side the project we had to find a suitable chatbot framework to incorporate into the website. Initial chat functionality was made using only HTML and CSS, as we had insufficient knowledge to design an aesthetic chat bot. The first variation was just a container that added a plain message when received from the server. There were 2 key aspects of the chat function that needed to be done. Firstly the design the chat button to be displayed in the front page, using tools we had learnt so far from previous terms eg. Illustrator / Photoshop we created our own custom button image and added transitions and effects to make the UX engaging. Secondly we needed additional chat features for the chat itself, namely the button to signal the send of a message and a clear distinct UI that separates the client and agent. After researching through several sources, we chanced upon BotUI that met our design expectation. It was a light-weight framework that was easy to import and had clear and simple documentation which did not take long for us to integrate into the webpage.

Testing Challenges

One of the challenges faced when conducting unit tests for the database was that the database used for queries was initially hardcoded to be the main app database. As such we had no way to specify that the newly created test database should be used when testing. We ended up adding a method to set a databaseName variable that would be used for all subsequent queries, and refactored all the database functions to refer to this variable.

For our integration tests, the main challenge faced was with testing functions without initializing connections with the Rainbow and Socket.IO APIs, which would require extensive setup and bear too much similarity to our system tests. This problem was partially solved by creating fake objects and functions and using the Sinon library to inject them into the functions. However, our functions involved some nested calls of other functions within the same module, and Sinon lacked the ability to replace objects within nested functions. Sinon also could not replace functions that were not attached to any object, so we could not replace the entire nested function with a fake. Finally, we settled on an ad hoc solution which was to refactor such that nested functions are called in the form of module.exports.myFunc, meaning that they will technically be attached to the exported module object and could then be replaced with Sinon.

In the UI test with Selenium, challenges we faced were mostly due to inexperience that we managed to overcome. Some included not being able to identify the correct element in our website that took awhile to solve. The problem arises due to the fact that we were using a light-weight external framework BotUI for the chat function. The BotUI did not not tag the elements in the DOM with a class or id, giving us a hard time trying to identify the elements we desired. After exploring all the options we had, we managed to find one that works, i.e., using the Xpath of the element instead of the class or id. Even though this solution is found, there were still limitations presented to us, this being the fact that even though we could identify the elements, when we wanted to incorporate our desired test cases, we found out that the Xpath of the element would change each time the webpage updates from user input. To mitigate this issue, we timed our delays in the test cases to ensure that at a particular time, an expected output should appear. Such implementation is found in the test for Flow 1, when the Client connects to the agent, a delay is added to wait for the response from the Server signifying that a connection is established with the message “You are now connected!”.Since adding of a delay sequentially would make the automation of the Test cumulatively slower when testing each department, there was an attempt to include threading to resolve this delay but threading would post another issue that we had to learn the hard way. When using threading to simulate a Client for each thread, what we found out was that by using threading, the Webdriver specific to that particular runs at approximately the same time as without threading. Meaning to say, even with multiple threads, the Webdriver controlling the elements of the webpage could only navigate a single thread at a time. This limitation is due to the Webdriver not being Thread-Safe, together with our knowledge we obtained in 50.003 ESC and online sources we deduced that the problem lies in browser bindings instantiated by the Webdriver.

Lessons Learnt

We followed a combination of the Agile and Iterative & Incremental development processes. One of the challenges faced when following this development process was with collaboration on code. Since we each had limited experience using GitHub, simultaneous development of the same codebase was a learning process. We learned to use the branching system in GitHub to separate our workspaces, however combining code was still mainly done by pulling the latest and adding in our own code before pushing it back up. Ideally we would like to learn to use pull requests to improve our workflow. Having different code styles was another issue we faced, and different styles would sometimes be present in the same file. Each member also had different ways of organizing their code into modules or functions. We should have decided on a coding standard before diving straight into coding.

Code should be written with the intention of being testable, especially for large projects. Code should be made more modular such that a test environment can be used if necessary, and hardcoding should be minimized. Functions with side effects such as making an API call can be modified to take in the API objects as parameters. This is to facilitate testing where faked versions of these objects can then be created and passed in to the functions.

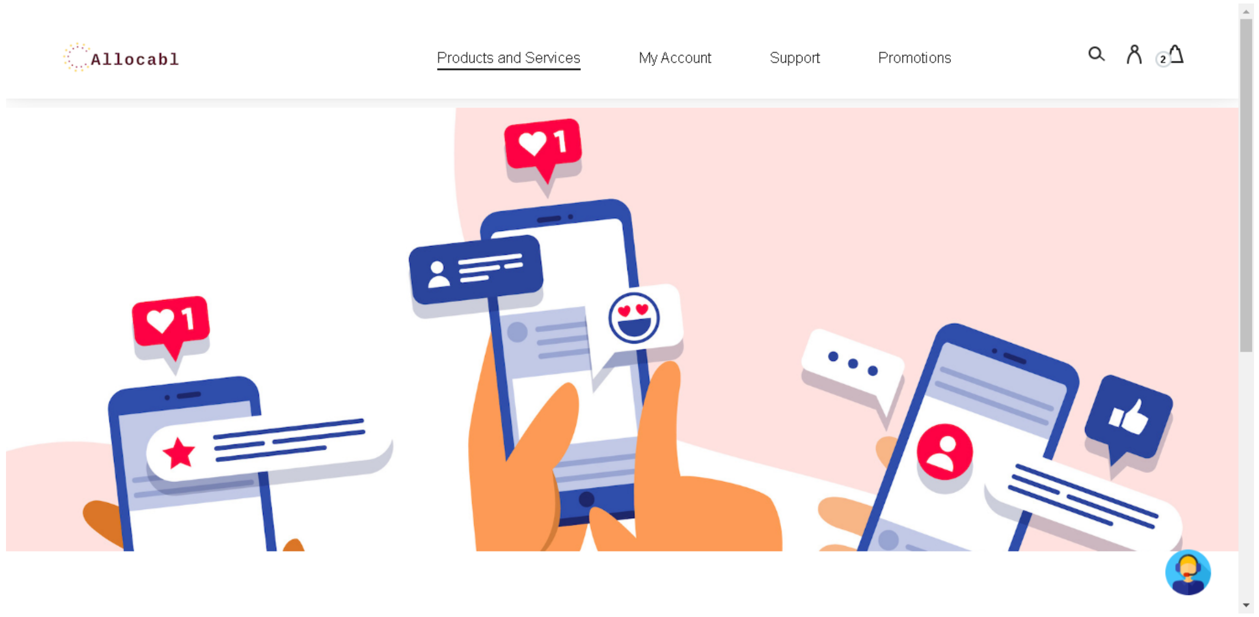
When designing modules to encapsulate functionality, the dependency tree between these modules should be considered in order to prevent circular dependencies. This involves careful consideration of the functionality that is assigned to each module. Shifting more objects into the parameters of a function as mentioned above could help with this as well, as it means that the objects do not need to be explicitly imported in the module itself and can be imported by the top level module and then passed in as a parameter.

From the start of the project, potential concurrency and thread-safety issues should be identified and considered. In our case, we were fortunate that the code was modular enough that adding synchronisation was a simple matter of adding locks to some key functions. However, this could reduce efficiency as the entire function is locked even when there are some parts of the function that might not be critical. If we had considered concurrency from the start, we might have been able to design our functions differently and only lock critical sections.

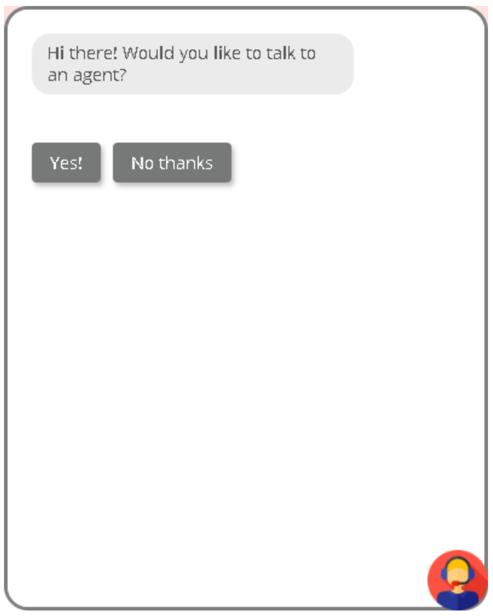
Deliverables

Github Link: https://github.com/EYLeong/Allocabl

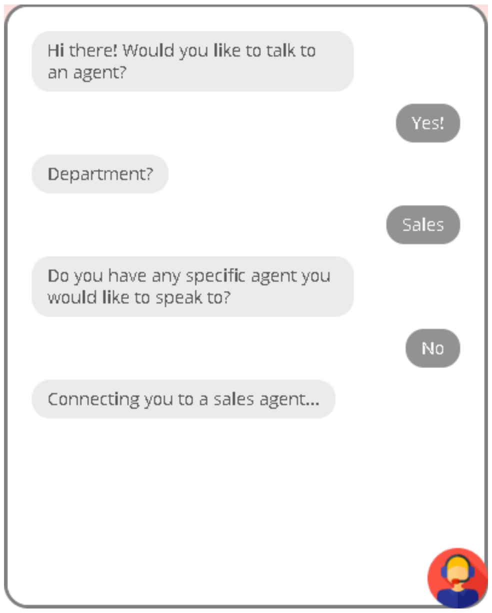
Screenshots



Homepage



Chat Prompt



Hi there! Would you like to talk to an agent?

Yes!

Department?

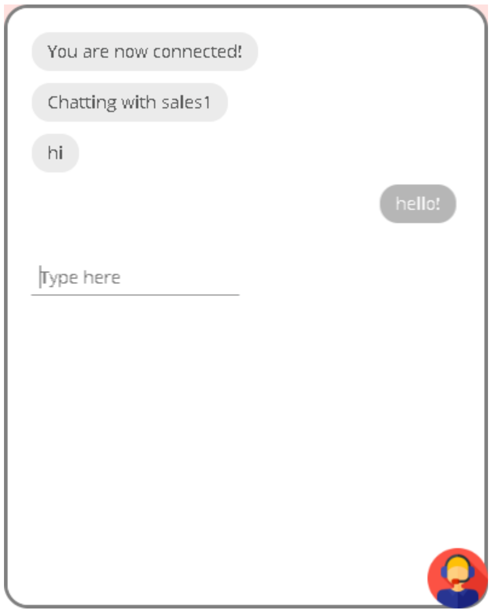
Sales

Do you have any specific agent you would like to speak to?

No

Connecting you to a sales agent...

Connecting to Agent



You are now connected!

Chatting with sales1

hi

hello!

Connected to Agent

Appendix A

Utils		
Function	**Behavior**	**Result**
isAlphaNum(string)	returns true if string is alphanumeric	Passed
	returns false if the string is non-alphanumeric	Passed
	returns false if string is empty	Passed
	returns false if parameter is not a string	Passed

Database-related Functions		
Function	**Behavior**	**Result**
setDatabase(databaseName)	throws an error if trying to set a database name that is not alphanumeric	Passed
	throws an error if trying to set an empty database name	Passed
	throws an error if trying to set a database name that is not a string	Passed
	sets the database for future queries to that specified by the given database name	Passed
getAgent(department)	returns available agents from specified department, ordered by customers served	Passed
	is empty when no agent is available	Passed
	is empty when the department does not exist	Passed
	is protected against sql injection	Passed

	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
checkAgentAvailable(department, agentID)	returns specified agent that is available	Passed
	is empty when specified agent is not available	Passed
	is empty when specified agent does not exist	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
getAgentDepartment(agentID)	returns department of agent	Passed
	is empty if agent is not found	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
getDepartment(socketID)	returns department of agent currently engaged to customerSocket	Passed
	is empty when given an invalid customerSocket	Passed
	is protected against sql injection	Passed

	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
incrementCustomersServed(agentID)	increments the number of customers served of an agent	Passed
	does nothing when given an invalid agent id	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
addSocketAgent(agentID, socketID)	adds a customerSocket to agent	Passed
	does nothing when given an invalid agent id	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
removeSocketAgent(agentID)	removes a customerSocket from corresponding agent	Passed
	does nothing if given an invalid socket id	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed

	throws an error if the parameters are not strings	Passed
setAgentAvailable(agentID)	sets an agent to available	Passed
	does nothing when given an invalid agent id	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
setAgentUnavailable(agentID)	sets an agent to unavailable	Passed
	does nothing when given an invalid agent id	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
setAgentOnline(agentID)	sets an agent to online	Passed
	does nothing when given an invalid agent id	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
setAgentOffline(agentID)	sets an agent to offline	Passed

	does nothing when given an invalid agent id	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
checkDepartmentOnline(department)	returns all online agents from specified department	Passed
	is empty when there are no online agents for the department	Passed
	is empty when the department does not exist	Passed
	is protected against sql injection	Passed
	throws an error if the database name is wrong	Passed
	throws an error if the parameters are not strings	Passed
addWaitlist(department, socketID)	adds sockets to waitlist in insertion order	Passed
	is protected against sql injection	Passed
	throws an error if the department does not exist	Passed
	throws an error if the department is not alphanumeric	Passed
	throws an error if the parameters are not strings	Passed
	throws an error if the database name is wrong	Passed

getFromWaitList(department)	returns the next customer in the waitlist for a department	Passed
	is empty if the waitlist is empty	Passed
	throws an error if the department does not exist	Passed
	throws an error if the department is not alphanumeric	Passed
	throws an error if the database name is wrong	Passed
removeFromWaitList(department)	removes the next customer from a waitlist by department	Passed
	does nothing if the waitlist is empty	Passed
	throws an error if the department does not exist	Passed
	throws an error if the department is not alphanumeric	Passed
	throws an error if the database name is wrong	Passed
removeFromWaitlistById(department, socketID)	removes a customer from the specified department waitlist	Passed
	does nothing if the customer is not in the waitlist	Passed
	is protected against sql injection	Passed
	throws an error if the department does not exist	Passed
	throws an error if the department is not alphanumeric	Passed

	throws an error if the parameters are not strings	Passed
	throws an error if the database name is wrong	Passed
clearDepartmentWaitlist(department)	clears the waitlist of a particular department	Passed
	throws an error if the department does not exist	Passed
	throws an error if the department is not alphanumeric	Passed
	throws an error if the database name is wrong	Passed
getDepartmentWaitlist(department)	returns the waitlist of a particular department	Passed
	is empty if the waitlist is empty	Passed
	throws an error if the department does not exist	Passed
	throws an error if the department is not alphanumeric	Passed
	throws an error if the database name is wrong	Passed
findSocketWaitlist(department, socketID)	looks for a socket ID in a given department's waitlist	Passed
	is empty if the entry is not found	Passed
	is protected against sql injection	Passed
	throws an error if the department does not exist	Passed

	throws an error if the department is not alphanumeric	Passed
	throws an error if the parameters are not strings	Passed
	throws an error if the database name is wrong	Passed
findSocketWaitlistDepartment(socketID)	returns the department of the waitlist that a given socket ID is in	Passed
	is protected against sql injection	Passed
	throws an error if the parameters are not strings	Passed

Appendix B

Socket Events		
Function	**Behavior**	**Result**
loginGuest(rainbowSDK, socket, inputs)	If an agent was available, picks the correct agent and sends the right data to the socket	Passed
	If an agent was available, sets agent to unavailable	Passed
	If an agent was available, increments the customers served of the agent	Passed
	If all agents are busy, sends the correct waitlist messages to the socket	Passed
	If all agents are busy, adds the socket ID to the correct department waitlist	Passed
	If no agent is online, sends the correct messages to the socket	Passed
updateClientsPositions(department)	notifies all sockets in waitlist about their positions	Passed
	does nothing if the waitlist is empty	Passed
checkWaitlist(rainbowSDK, department)	does nothing if waitlist is empty	Passed
	If the waitlist is not empty, notifies socket that an agent is available	Passed
	If the waitlist is not empty, calls loginGuest with the right parameters	Passed
	If the waitlist is not empty, calls updateClientsPositions with the right parameters	Passed

disconnect(rainbowSDK, socket)	If the customer is talking to an agent, removes the socket ID from the agent and sets agent to available	Passed
	If the customer is talking to an agent, calls checkWaitlist with the right parameters	Passed
	If the customer is not talking to an agent, does nothing if not in waitlist	Passed
	If the customer is not talking to an agent, removes socket ID from waitlist	Passed
	If the customer is not talking to an agent, calls updateClientsPositions with the right parameters	Passed

Rainbow Events		
Function	**Behavior**	**Result**
onAgentStatusChange(rainbowSDK, id, presence)	If an agent comes online, sets agent to online and available	Passed
	If an agent comes online, calls checkWaitlist with the correct parameters	Passed
	If an agent goes offline, sets agent to offline and unavailable	Passed
	If an agent goes offline, notifies all affected waitlistees	Passed

Should we have a limit to the number of customers that can be in the waitlist?

Final Report
Implementation Challenges (Daniel/Varsha - Frontend, EnYi/Ling Tian - Backend)

1) Algorithmic (might be specific to a few projects only (e.g., for games))
2) Engineering (e.g., usage of certain tools, integration issues)
3) Testing

Update testing - Show test inputs and results (Daniel/Varsha - system, EnYi/Ling Tian - unit)
Lessons learnt (Daniel/Varsha - Frontend, EnYi/Ling Tian - Backend)

1) What software development process was followed?
2) Was there any challenges faced in following the process?
3) What are the lessons learnt in general after finishing the project?