Final Iteration Submission:

CabX
Clyde Bazile (cb3150),
Benedikt Schifferer (bds2141),
Xiao Lim (xl2669),
Adiza Sumuna Awwal (asa2201)

Github Link: https://github.com/bazile-clyde/CabX

1. **Use cases:**

CABX-001 Use Case:

Title	Given an origin and destination, find ride-hailing service
Actor	Member (Registered User)
Description	The main component of the CabX app. The member inputs an origin and destination address to find valid ride-hailing services around them. We fixed an issue in the second iteration where the app arbitrarily chose a location when the input has multiple possibilities (e.g. Chipotle). Now, suggestions will appear while typing so the user will know the address they selected is the intended one. Visually, we've also implemented a map that displays the user's current location and the route to the selected destination.
Basic Flow	1. The system provides input areas for the user to specify the origin and destination address 2. The member fills in the origin and destination address 3. System shows potential location suggestions while member types in the addresses 4. The member clicks on the search button 5. The system returns available ride-hailing services, the price of each service and the estimated time it takes to arrive at the destination
Alternate Flow	2A. The member realizes that they accidentally switched the origin and destination addresses. 1. The member clicks the switch button to reverse the origin and destination addresses 2. The use case returns to step 3 and continues

CABX-002 Use Case:

Title	Sort by Price
Actor	Member (Registered User)
Description	After getting a list of ride-hailing services, the member wants to simplify their decision-making by sorting the list by price (lowest to highest)
Basic Flow	1. System shows member the list of ride-hailing services 2. Member clicks on the “money” symbol to sort by price 3. System returns the list of ride-hailing services, now sorted by lowest to higher estimated price
Alternate Flow	N/A

CABX-003 Use Case:

Title	Sort by ETA
Actor	Member (Registered User)
Description	After getting a list of ride-hailing services, the member wants to simplify their decision-making by sorting the list by estimated time it’ll take to arrive at their destination (shortest to longest)
Basic Flow	1. System shows member the list of ride-hailing services 2. Member clicks on the “time” symbol to sort by ETA 3. System returns the list of ride-hailing services, now sorted by estimated time to arrive at destination (shortest to longest)
Alternate Flow	N/A

CABX-004 Use Case:

Title	Login to Access History
Actor	CabX User (New or Returning)
Description	When opening the app, ~~a registered user will be prompted by the system to login to an existing account using previously registered~~

	~~credentials. A new user will have to create an account by providing an email address and password~~ users will encounter the login page, with the option to sign up for a new account. If a user clicks “sign up”, an additional “confirm password” text area will appear. When a user signs up for an account, they will receive an email to verify their account. A user can only log in after confirming their credentials by clicking on the verification link in their email. Users who forgot their password can also reset their password by clicking on the “forgot my password” button.
Basic Flow	1. User opens the app 2. System shows user login page 3. User inputs email address and password 4. User clicks “sign in” 5. System validates the email address and password are a valid combination 6. System redirects user to main page of the app (CABX-001)
Alternate Flow	2A. User does not have existing account 1. User navigates to sign up page 2. System prompts user to create an account by entering an email address and password and confirm the password 3. User clicks “create account” 4. User verifies account via email link 5. System checks if ~~the email address is valid and if the password fits the criteria~~ user has verified account 6. System stores email address and password combination 7. System redirects user to login page 8. The use case returns to step 3 and continues 2AA. Both passwords do not match when user is creating an account 1. User enters incorrect matching password 2. System shows an error when validating passwords and prompts user to re-enter password 3. The use case returns to step 2 of 2A 3A. User inputs incorrect email address 1. System returns error when validating email address 2. System shows user a warning and prompts them to re enter email address 3. The use case returns to step 3 and continues 3B. User inputs incorrect password 1. System returns error when matching password to email address

	2. System shows user a warning and prompts them to reenter password 3. The use case returns to step 3 and continues 3C. User has forgotten ther password 1. User inputs email address 2. User receives an email prompting them to click on a link to reset password 3. User enters new password and saves it 4. The use case returns to step 3 and continues

CABX-006 Use Case:

Title	Save Previous Search History
Actor	Member (Registered User)
Description	Rather than having users retype their previous search queries, the app saves them in the user's profile data. The search queries will be accessible by clicking the button next to the search box. Search queries are now saved in the MongoDB database and we fixed the issue of duplicates in recent queries.
Basic Flow	1. Member opens the app 2. Member clicks on the icon next to origin or destination input area 3. The app displays a list of the member's five most recent search queries 4. Member clicks one of their five most recent search queries and have it populate the search field 5. System redirects member to main app with search field populated
Alternate Flow	4A. Member doesn't find relevant search in history 1. Member exits the search query page by clicking on the back button to return to the main app 2. The use case returns to step 5, but search field remains the same as before

2. **Test plan:**

Github link: https://github.com/bazile-clyde/CabX/tree/master/backend/test

In general, the backend tests covers each function with a separate number of test cases, encompassing the important cases.

High level functions handle two cases:

1. Successful responses from low level function
2. Error response from low level function

The low level functions have more test cases with specific edge cases and high level functions have simpler test cases.

To test functions that call APIs, we create mock json responses (using nock) that the API would have produced. We create four different mock json responses:

1. Successful response
2. Error, aka a response code that's not 200
3. Delayed response (1499ms) (boundary condition)
4. Timeout (1501ms) (boundary condition)

Since we're creating mock API responses, different input values to the APIs are not tested. If the parameter for an API is not valid, the API will return empty or a response code that isn't 200.

If a function has a loop, there are success cases with 0, 1, 2 and 3 iterations.

Testing getFindRides.js with getFindRides_test.js

function: findRides(sAddressFrom, sAddressTo)

Equivalence classes:

- Success — sAddressFrom and aAddressTo are found and returns the expected json array
- sAddressFrom has an error:
    - To mock Bing's API, we return a 500 response code
    - Bing could not find sAddressFrom (empty response)
- sAddressTo has an error:
    - To mock Bing's API, we return a 500 response code
    - Bing could not find sAddressTo (empty response)

Testing getLatLongFromAddress.js with getLatLongFromAddress_test.js

function: requestLatLongFromAddress(sAddress)

Mock of Bing API - equivalence classes:

1. Success response

   a. Bing API returns response code 200
   b. requestLatLongFromAddress returns json object and tests if specific fields are included
2. Error code
   a. Bing API returns response code 500 (or any other code than 200)
   b. requestLatLongFromAddress rejects with error
3. Long response time: Success response with 1499ms response time
   a. Bing API returns response code 200 but takes 1499ms
   b. requestLatLongFromAddress returns json object and tests if specific fields are included
4. Timeout: Success response with 1501ms response time
   a. Bing API returns response code 200 but takes more than 1501ms
   b. requestLatLongFromAddress rejects with error ESOCKETTIMEDOUT

Boundary conditions:

- Delayed response: 1499ms
- Timeout: 1501ms

function: processBingLatLongResponse(jsonResponses)

Equivalence classes/boundary conditions:

1. Valid json object
2. Invalid json object:
   a. Missing resourceSets field
   b. Missing resources field

function: processBingResponseBody(jsonResponsesBody)

valid json object: correct field names

Equivalence classes/boundary conditions:

1. Array of json objects with 1 valid object (correct fields) (loop 1 time)
2. Array of json objects with 2 valid objects (correct fields) (loop 2 times)
3. Array of json objects with 3 valid objects (correct fields) (loop 3 times)
4. Empty array (loop 0 times)
5. Array of json objects: one valid object, one invalid object
6. Array of json objects: two invalid objects

function: getRelevantInformationFromLatLongResponse(jsonResponseBody)

valid json object: correct field names

Equivalence classes/boundary conditions:

1. Valid json object
2. Invalid json object:
    a. Missing name field
    b. Missing confidence field
    c. Missing geocodePoints field
    d. Missing address field
    e. Missing coordinates field
    f. Only one coordinate in array

Testing getUber.js with getUber_test.js

function: uberPrices(fromLat, fromLong, toLat, toLong)

Mock of Uber API - equivalence classes:

1. Success response
    a. Uber API returns response code 200
    b. uberPrices returns mocked json response. The unit test compares the mocked json response with the real json response to see if they are the same
2. Error code
    a. Uber API response code 500 (any other code than 200)
    b. uberPrices rejects with error
3. Long response time: Success response with 1499ms response time
    a. Uber API returns response code 200 but takes 1499ms
    b. uberPrices returns mocked json response. The unit test compares the mocked json response with the real json response to see if they are the same
4. Timeout: Success response with 1501ms response time
    a. Uber API returns response code 200 but takes more than 1501ms
    b. uberPrices rejects with error ESOCKETTIMEDOUT

Boundary conditions:

- Delayed response: 1499ms
- Timeout: 1501ms

function: processUberResponseBody(jsonResponsesBody)

valid json object: correct field names

Equivalence classes/boundary conditions:

1. Array of json objects with 1 valid object (correct fields) (loop 1 time)
2. Array of json objects with 2 valid objects (correct fields) (loop 2 times)
3. Array of json objects with 3 valid objects (correct fields) (loop 3 times)
4. Empty array (loop 0 times)

5. Array of json objects: one valid object, one invalid object
6. Array of json objects: two invalid objects

function: getRelevantInformationFromUberResponse(jsonResponseBody)

valid json object: correct field names

Equivalence classes/boundary conditions:

1. Valid json object
2. Invalid json object:
    a. Missing display_name field
    b. Missing distance field
    c. Missing high_estimate field
    d. Missing low_estimate field
    e. Missing duration field

Testing getLyft.js with getLyft_test.js

function: lyftPrices(fromLat, fromLong, toLat, toLong)

Mock of Lyft API - equivalence classes:

1. Success response
    a. Lyft API returns response code 200
    b. lyftPrices returns mocked json response. The unit test compares the mocked json response with the real json response to see if they are the same
2. Error code
    a. Lyft API returns response code 500 (any other code than 200)
    b. lyftPrices rejects with error
3. Long response time: Success response with 1499ms response time
    a. Lyft API returns response code 200 but takes 1499ms
    b. lyftPrices returns mocked json response. The unit test compares the mocked json response with the real json response to see if they are the same
4. Timeout: Success response with 1501ms response time
    a. Lyft API returns response code 200 but takes more than 1501ms
    b. lyftPrices rejects with error ESOCKETTIMEDOUT

Boundary conditions:

- Delayed response: 1499ms
- Timeout: 1501ms

function: processLyftResponseBody(jsonResponsesBody)

valid json object: correct field names

equivalence classes/boundary conditions:

1. Array of json objects with 1 valid object (correct fields) (loop 1 time)
2. Array of json objects with 2 valid objects (correct fields) (loop 2 times)
3. Array of json objects with 3 valid objects (correct fields) (loop 3 times)
4. Empty array (loop 0 times)
5. Array of json objects: one valid object, one invalid object
6. Array of json objects: two invalid objects

function: getRelevantInformationFromLyftResponse(jsonResponseBody)

valid json object: correct field names

equivalence classes/boundary conditions:

1. Valid json object
2. Invalid json object:
    a. Missing display_name field
    b. Missing estimated_distance_miles field
    c. Missing estimated_cost_cents_max field
    d. Missing estimated_cost_cents_min field
    e. Missing estimated_duration_seconds field

3. **Branch coverage:**

Automated test coverage reports:
https://github.com/bazile-clyde/CabX/blob/master/reports/159/report_coverage.txt

Currently, there are a few reports. The automated push from travis CI broke after the separate branch was behind the master. Now, travis-ci pushes to the master branch.

We use istanbul for test coverage reports:

```
./node_modules/.bin/istanbul cover ./node_modules/.bin/_mocha
frontend/test/*.js backend/test/*.js && ./node_modules/.bin/istanbul
check-coverage --statements 70 --functions 70 --lines 70
```

```
=============================== Coverage summary ==========================
Statements   : 90.45% ( 142/157 )
Branches     : 83.82% ( 57/68 )
Functions    : 91.3% ( 21/23 )
Lines        : 90.45% ( 142/157 )
```

```
==========================================================================
```

Branch coverage definition: https://github.com/dwyl/learn-istanbul
Conditional statements create branches of code which may not be executed (e.g. if/else). This metric tells you how many of your branches have been executed.

How Istanbul calculates branch coverage:
https://stackoverflow.com/questions/26618243/how-do-i-read-an-istanbul-coverage-report
Has each branch (also called DD-path) of each control structure (such as if and case statements) been executed? An example is: given an if statement, have both the true and false branches been executed?

Exploring the behavior of branch coverage with Istanbul shows the following pattern:

- Istanbul counts all branches (e.g. if-else statements) and checks if each branch was executed at least once
- Istanbul does not count different permutations executing the different branches

Example 1:

```
function() {
      if(variable) {
            //code
      }
      return(something)
}
```

Istanbul counts example 1 as 2 branches - if and else branch

Example 2:

```
function() {
      if(variable1) {
            //code
      } else  {
            //code
      }
      if(variable2) {
            //code
      } else  {
            //code
      }
      if(variable3) {
            //code
      } else  {
            //code
      }
      return(something)
}
```

Istanbul counts example 2 as 6 branches. However, there are 2^3 = 8 possible ways to execute the function.

Test cases for loop initiation:
Excerpt from the 16 October lecture —
“Loop statements are multi-way branches, perhaps ∞ many, but we need to stop somewhere so typical rule of thumb is zero times, one times, more than one time through loop”

There are three functions using for-loops:
- **processUberResponseBody(jsonResponsesBody)**
- **processLyftResponseBody(jsonResponsesBody)**
- **processBingLatLongResponse(jsonResponses)**

Each of them is covered with 4 test cases related to loop iterations. The for-loop is executed 0, 1, 2 and 3 times.