Test Case Spec (QA)

Test Strategy & Approach

Our approach to the test strategy phase relies on a hybrid of black (functional and validation) and white (unit and integration) box testing. An integral part of the mobile UI development cycle consists of heavy unit testing. We are concerned with the optimal interface usability ("look and feel") of the product as well as the acquisition of data. Unit tests have helped us experience the intended user interface design and also to see if it was possible to capture data associated with web and blockchain APIs. The success of the unit testings could only be made possible from the integrity of the source code; it had to be devoid of errors and contain an efficient logical design to carry out its intended functions.

Our standard toward integration testing was straightforward in the sense that we used a top-down approach in uncovering errors related to our interface components. Our main source of control and dependencies were the actual android devices/emulators that we used to test our web API queries. As with most development processes, new features were immediately tested for and its validity was determined by its completeness and conformity (functional tests).

Validation and system tests were required from various emulators to ensure that a consistency existed between different platforms. The IDE used for the development of the mobile UI was conducted on Android Studio.

Test Plan

The test plan structure accommodates for the testing of both Android Studio devices and emulators. The test plan is broken down categorically as follows:

System tests - conducted through seven different types of emulators listed in α | β | γ notation where α represents the model name, β is the dimensions and dpi setting, and γ represents the API level; all emulators ran on an x86 CPU architecture. One physical Android device that was used for our testing purposes was done on a Nexus 6.

***Emulators*:**

Pixel API Q | 1080 x 1920: 420 dpi | Q
Pixel 3 XL | 1440 x 2960: 560 dpi | 22
2.7 QVGA | 240 x 320: ldpi | 22
Pixel 3 XL | 1440 x 2960: 560 dpi | Q
Galaxy Nexus| 720 x 1280: xhdpi | 22
Nexus One | 480 x 800: dpi | 22
Nexus 6 | 1440 x 2560: 560 dpi | 22

Unit tests (source code) - (focuses on logic and data structures)*Test Cases* category contains an extensive list of scenarios. To ensure that the intended actions were translated to source code and that the integrity of the logic was not compromised.

Integration tests - new components that are introduced are given test cases. Our emulators/devices are the main modules that determined the outcome of our working set model. For every new component that we test for, we define a set of goals or expected outcomes. Any bugs that we encounter automatically introduces a new priority to our integration testing. The quicker we solve the problem, the quicker we may move onto the next component.

Functional tests - ensures that each software function or feature is tested for completeness and conformity. Will the function/feature do its assigned job? For example, will the login text edit display input from a user? Are the UI buttons in accordance with its function?

Validation tests - Android Studio prints to the IDE console via Logcat, main uses of Logcat will help determine if expected data gathered from web API or blockchain is valid. Postman is a third-party tool that helps us verify the proper format of data. We use Postman's response as a debugging tool to see if our retrieved data matches the intended format.

Test Cases & Requirement Identifiers

Each test case contains a unique identification number and a description, the same applies to requirements. Our test cases focus on compatibility since each test case is a different device/emulator. REQ12 strictly applies to the HyperLedger blockchain and not the other blockchains since permission requests were only granted due to availability. Resources from other groups were only offered if they could afford the expense, thus we did not try to ask from all blockchain groups.

Test Case ID	Test Case desc.	Requirement ID	Requirement desc.
TC1	Pixel	REQ1	application starts up on login screen
TC2	Pixel 3 XL (Q)	REQ2	text field accepts user input on login page
TC3	Pixel 3 XL (22)	REQ3	login to application
TC4	2.7 QVGA	REQ4	text field accepts user input on product lookup page

TC5	Galaxy Nexus	REQ5	layout for all pages are displayed accordingly to IDE standards
TC6	Nexus One	REQ6	back button pressed from search results page sends user to previous page
TC7	Nexus 6	REQ7	search results page is scrollable
TC8	Nexus 6 (device)	REQ8	access restful web API (Heroku App) data prior to blockchain
		REQ9	access (HyperLedger) blockchain data
		REQ10	access (Etherium) blockchain data
		REQ11	access (Open Chain) blockchain data
		REQ12	Parse data info from blockchain (Hyper Ledger) onto Android Studio's Logcat (JSON objects)

**Only one physical Android device (labeled: device) is used in the testing process, the rest are emulators.*

Traceability Matrix

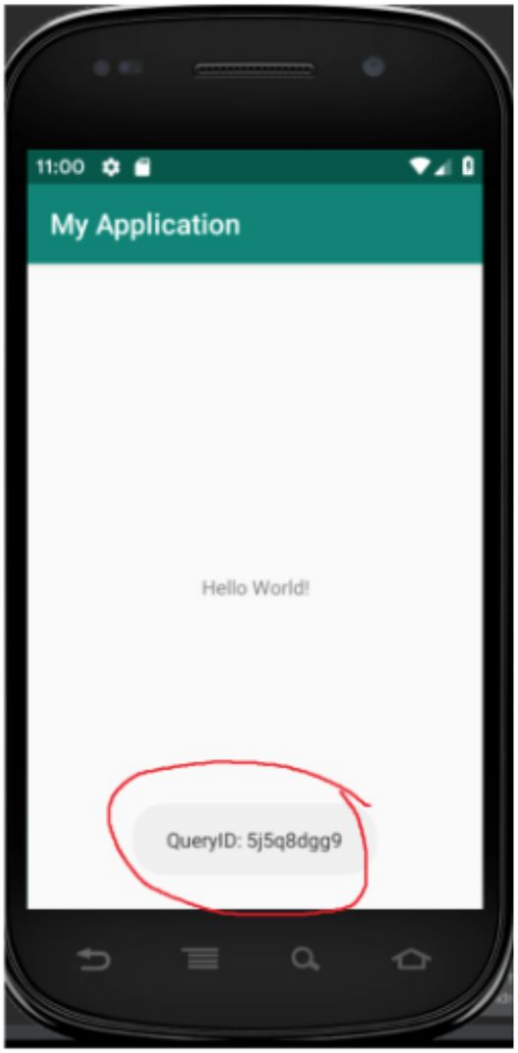
	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	**Requirement Identifiers**	Reqs tested	**REQ1**	**REQ2**	**REQ3**	**REQ4**	**REQ5**	**REQ6**	**REQ7**	**REQ8**	**REQ9**	**REQ10**	**REQ11**	**REQ12**
2	Test Cases	96	8	8	8	8	8	8	8	8	8	0	0	8
3	**TC1**	10	x	x	x	x	x	x	x	x	x			x
4	**TC2**	10	x	x	x	x	x	x	x	x	x			x
5	**TC3**	10	x	x	x	x	x	x	x	x	x			x
6	**TC4**	10	x	x	x	x	x	x	x	x	x			x
7	**TC5**	10	x	x	x	x	x	x	x	x	x			x
8	**TC6**	10	x	x	x	x	x	x	x	x	x			x
9	**TC7**	10	x	x	x	x	x	x	x	x	x			x
10	**TC8**	10	x	x	x	x	x	x	x	x	x			x

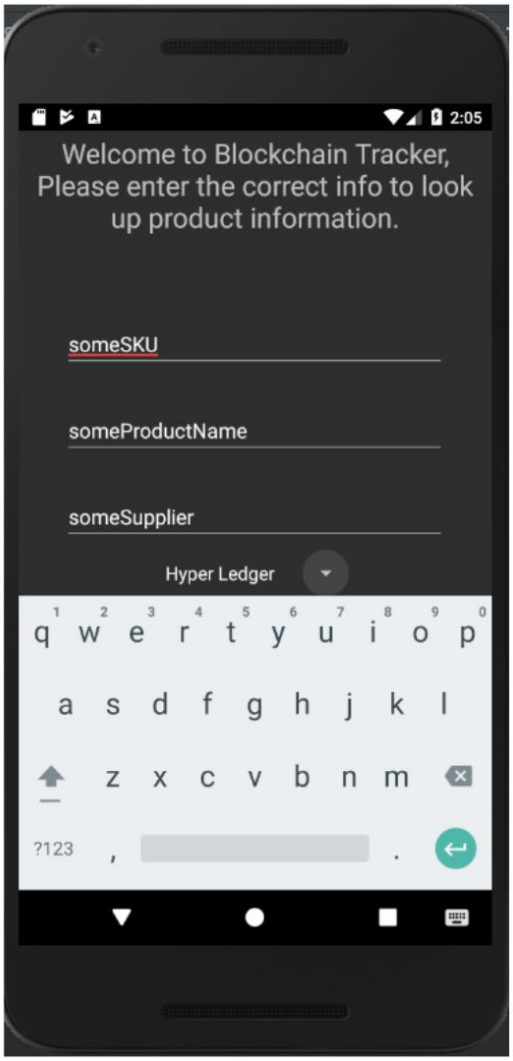
Test Results Summary

The failed cases corresponding to REQ10 and REQ11 is involved with accessing blockchain data from Etherium and Open Chain (refer to the *Test Cases & Requirement Identifiers* for detailed definitions). The blockchain was configured and set up in a narrow time-window prior to our class demonstration, therefore, we had difficulties in trying to access all blockchain data. We had to contact the groups responsible for providing the blockchain services and request that they have the blockchain in active operation. We understood that our request might pose a problem for some groups, for the process of keeping the blockchain active could be a relatively expensive task. However, we communicated with the HyperLedger blockchain group and they were willing to keep their blockchain active for our testing purposes.

To expedite the testing process, we by-passed the process of inputting username and password to log into our mobile application. We wanted to navigate the mobile UI without restrictions to check up on our acquired data. We were successful in running our application across all devices and emulators with little issues. The following screenshots highlight some of our main testing processes for our mobile UI development.

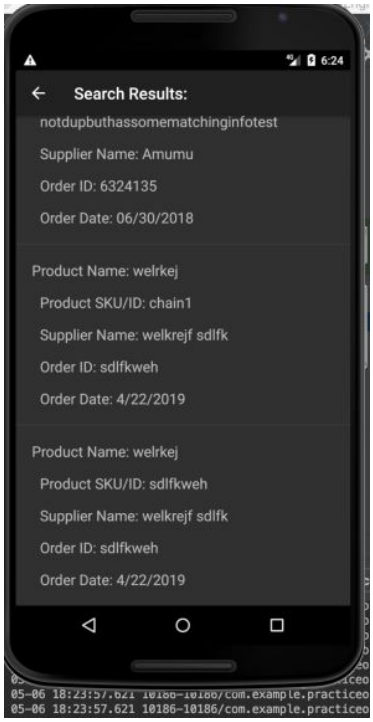
In the beginning stages of our development, the blockchain groups had not deployed their services, therefore we had to use the web API query information as a test case. Our unit test below shows that we post to the API and retrieve a query id.





The web API team had to make some changes to accommodate for the blockchain groups' last minute changes. As a result, the sample query was taken down and made inaccessible prior to the class demonstration. This is the results of acquiring the sample query from the web API prior to the changes. A reading from our Logcat displays the relevant data that's expected from postman.

```
2019-05-06 12:13:53.750 28343-28373/com.example.practiceone D/LisaAPIConnectionTest: Name: name0, SKU: 0, Supplier: prevownername0, OrderID: , OrderDate:
2019-05-06 12:13:53.750 28343-28373/com.example.practiceone D/LisaAPIConnectionTest: Name: name1, SKU: 1, Supplier: prevownername1, OrderID: , OrderDate:
2019-05-06 12:13:53.750 28343-28373/com.example.practiceone D/LisaAPIConnectionTest: Name: name2, SKU: 2, Supplier: prevownername2, OrderID: , OrderDate:
```



We had to provide our own JSON objects to test for alternative case scenarios. We wanted to work with data information that was different compared to the sample data. We created blockchain and component queries in the form of nested JSON objects that contained JSON arrays (see following pages).

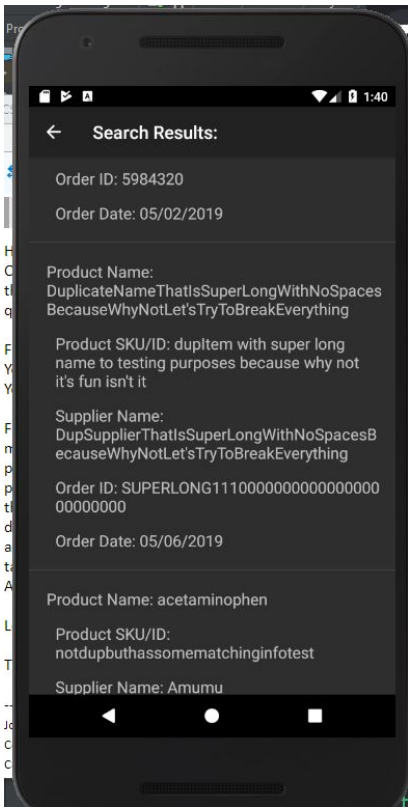

```
{
    "status": "ok",
    "data": [
        {
            "productID": "6",
            "productName": "xanax",
            "owner": "Yasuo",
            "previousOwner": "Annie",
            "quantity": "120",
            "pricePerUnit": "2",
            "totalPrice": "6",
            "orderID": "5984320",
            "transactionDate": "05/02/2019"
        },
        {
            "productID": "1",
            "productName": "morphine",
            "owner": "Annie",
            "previousOwner": "Bard",
            "quantity": "55",
            "pricePerUnit": "4",
            "totalPrice": "12",
            "orderID": "1254314",
            "transactionDate": "02/12/2019"
        },
        {
            "productID": "2",
```



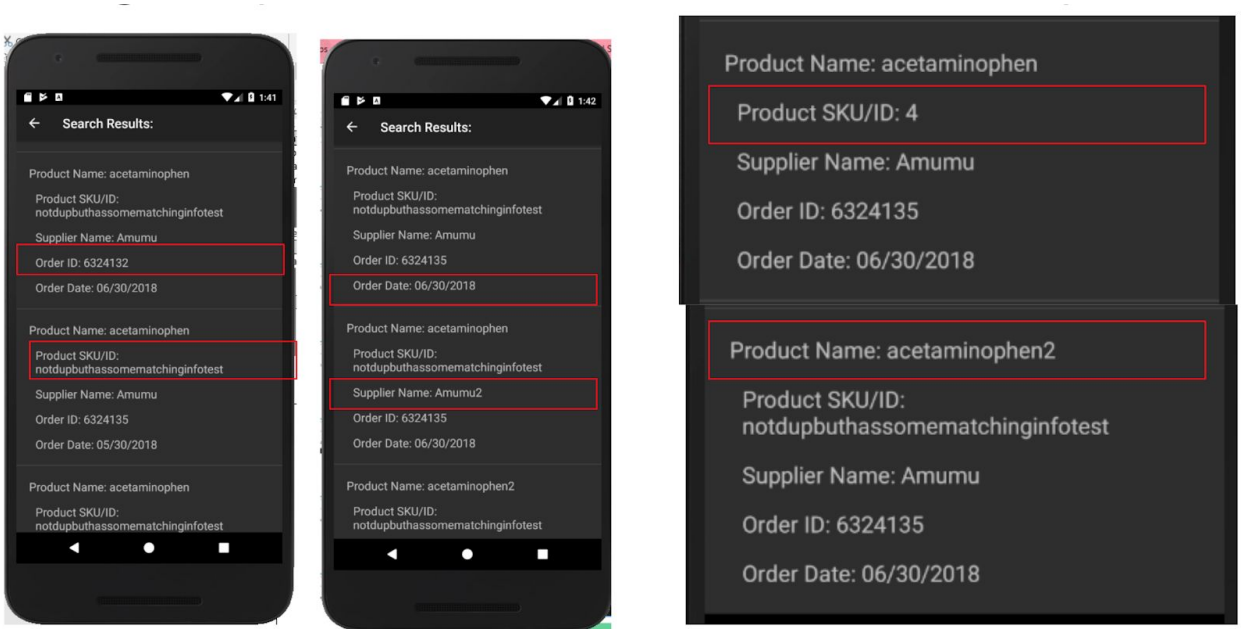
```
{
    "status": "ok",
    "data": [
        {
            "productID": 1,
            "previousOwner": "Vermont_Medical_Supply",
            "productName": "Xanax-1"
        },
        {
            "productID": 2,
            "previousOwner": "Pico_Medical_Supply",
            "productName": "Xanax-1"
        },
        {
            "productID": 3,
            "previousOwner": "Pomona_Medical_Supply",
            "productName": "Xanax-1"
        },
        {
            "productID": 4,
            "previousOwner": "Hometown_Medical_Supply",
            "productName": "Xanax-2"
        },
        {
            "productID": 1,
```

An important data validation to test for was the ability to see long strings and duplicate items on the mobile UI.

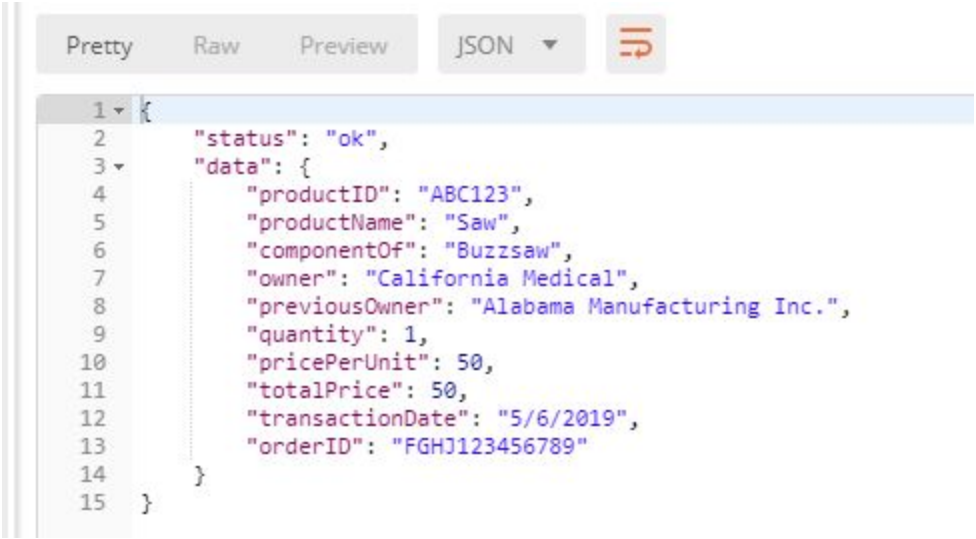

```
{
    "productID": "dupItem with super long name to testing purposes because why not it's fun isn't it",
    "productName": "DuplicateNameThatIsSuperLongWithNoSpacesBecauseWhyNotLet'sTryToBreakEverything",
    "owner": "DupOwner",
    "previousOwner": "DupSupplierThatIsSuperLongWithNoSpacesBecauseWhyNotLet'sTryToBreakEverything",
    "quantity": "1700000",
    "pricePerUnit": "4000000",
    "totalPrice": "12",
    "orderID": "SUPERLONG11100000000000000000000000",
    "transactionDate": "05/06/2019"
},
{
    "productID": "dupItem with super long name to testing purposes because why not it's fun isn't it",
    "productName": "DuplicateNameThatIsSuperLongWithNoSpacesBecauseWhyNotLet'sTryToBreakEverything",
    "owner": "DupOwner",
    "previousOwner": "DupSupplierThatIsSuperLongWithNoSpacesBecauseWhyNotLet'sTryToBreakEverything",
    "quantity": "17",
    "pricePerUnit": "4000000",
    "totalPrice": "12",
    "orderID": "SUPERLONG11100000000000000000000000",
    "transactionDate": "05/06/2019"
}
```

Testing unique items with near similar descriptions was also on our test case agenda:



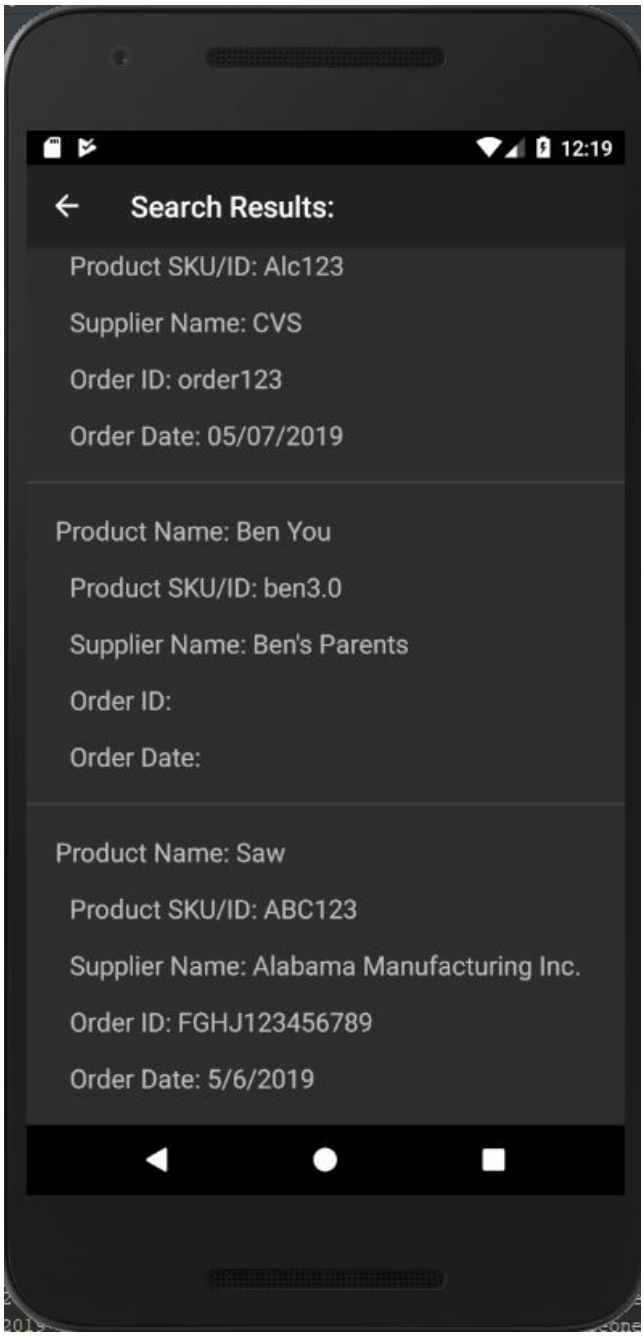
The most important test that was conducted involved in acquiring data from the HyperLedger blockchain. We had to use postman to verify the response data and then simply retrieve and display the information on our mobile UI.



**Postman verifies how the retrieved data should look like.*

**A unit test that existed outside of the main UI program displaying retrieved data on Android Studio's console (Logcat).*

**Unit testing within the main UI program to confirm the retrieval of HyperLedger's data.*



**Display of blockchain data on mobile UI.*