Abstract

The project aims at reducing the queue waiting-time of customers in large supermarkets such as Tesco. Upon examining and talking to customers from various supermarket outlets, it is evident that long queues lead to customer dissatisfaction. Shoppers tend to overestimate the waiting time leading to a negative perception of the overall queuing time. QwikScan that supports self-scanning and checkout, reduces queuing time leading to higher customer satisfaction. Tesco customers can scan the products as they buy, on their own phone using the app and make an online payment, thus eliminating their need to wait in the long queues. Customers also have the alternative option of generating a single QR code of total products and pay at the counter. Tesco already has self-checkout kiosks but they are limited to few numbers and, also stationery. Our app makes every user's phone a kiosk and decreases queue time by a big margin.

Description of two similar apps

1. QThru

The QThru app enables customer to scan items as they shop by using their smartphone and later check-out by scanning a kiosk QR code.

2. Mobile Checkout

The Mobile Checkout app debuted by FutureProof. The app recognizes the item and the shopper can then weigh it on special scales that FutureProof deploys in the store. The user then scans the displayed weight and the final price appears as a line item in the app. The customers make payment via debit or credit card.

How QwikScan differs from above apps

The motive of both apps is to provide simplicity to the shopping experience but in the case of QThru, scanning the kiosk QR code is the only possible method to make payment which still results in a queue and inconveniences the customers who wish to pay online. On the other hand, the Mobile Checkout app only increases the customers' wait time as they have to scan twice; the products and the weight.

Literature Research

Barcode Scanner:

To scan the products, an open source Scanner called ZXing (Zebra Crossing) is used which allows a user to scan 1-D or 2-D graphical barcodes with the camera on their Android device. The program turns that scan into the original data that is represented by the barcode. This allows the user to get to web addresses, geographical coordinates, small pieces of text, and much more just by pointing their device's camera at a barcode. The program is added as a JAR dependency inside the project.By using the ZXing library, users don't need to worry if a barcode scanner is not installed, because the integration

classes provided will take care of this. By importing the ZXing integration classes into the app, the user scans easier.

Android Pay:

Android pay works by using a method called “Tokenization”. When a user adds a card to her Android pay account, a token is generated. This token does not contain card details but instead holds an encrypted key that is used by user’s supporting bank to verify that the card used is indeed valid. The teller's machine prompts the customer/user to touch their phone off of the wireless payment terminal to start the transaction. Once the device is touched off of the teller’s wireless terminal, the token is passed to the merchant’s bank, who then passes it to the TSP (Token Service Provider). The TSP verifies the token and then passes the customer’s credit card details to the payment network consisting of the card provider i.e. visa, MasterCard etc. and the customer's bank branch. The bank is given the card details to verify and authorize the payment using the card. This message is then passed back through the same network and confirms to the teller that the transaction was either successful or that it didn’t work. The greatest part of android pay is that since the service doesn’t require any proprietary hardware for the wireless transfer, normal standard wireless car terminals can be used as all of the processing is done in the backend which doesn’t inconvenience the merchant in any way, helping the possibility for widespread integration.

App Originality

1. The app’s design is very loosely-coupled with supermarket’s servers and so it can be customized for any store by minimal design and database change. The app acts as a middleware and can be reused.
2. The interface is user friendly and simplistic. The app has a very basic end-to-end flow without many branches and inter-dependencies.
3. Unlike other similar apps, this app provides multiple payment options. Apart from the online payment option, it also has a QR code generation mode. Anyone who is not comfortable paying online or does not possess a Credit/Debit card can afford to generate a QR code at the end of their products scanning. This will give them a single QR code (it will be synced with Tesco’s server) which they can show at the counter and make a traditional cash payment. This still reduces the queue time because the shop scans a single accumulated code instead of scanning multiple; say 50, and individual products.
4. The app also determines the user’s location and automatically lists the nearest Tesco shops available using Google Maps. This can be used by customers who travel a lot and want to find and shop on the fly. With all features built under one platform, the app acts as a one stop solution.

Key Design consideration

The overall key design consideration of the app was to make it as intuitive as possible, at no point is the user unsure as to right path to find what they want. This was successfully implemented by keeping the functionality and design very linear and sequential. Another key point was to make sure the app held a consistent styling/look across all smartphone screens size, this was tested on phones starting from 3.7-inch screens all the way up to 5.7-inch screens.

When coming up with the concept of the app, emphasis was put on the target users of the app, who are they? Age? Job? Income? And after some research on the market that the app would be correlated to primary and secondary target users were identified.

The primary target users are:

- Age between 21 – 38
- Income 30,000+
- International
- Graduate
- Single or Married

The secondary target users are:

- Age between 18 - 36
- Income 10,000+
- International
- Student
- Single

The key use cases consist of (screenshots can be seen in appendix):

- User logs in -> Starts Scanning -> Finishes Scanning -> Chooses Payment Option -> Pays
- User logs in -> Starts Scanning -> Finds Nearest Store with Map -> Starts Scanning -> Finishes Scanning -> Chooses Payment Option -> Pay

App Implementation

Below is a short explanation of different activities in the app, the goals they achieve and the Android features used to achieve them

Login - This activity uses the Google Sign-In API to enable users of the app to log in using their Google Accounts. A check is done to determine if there is a cached sign-in result object. A GoogleApiClient class object is needed to access the GoogleSignIn API. When the user clicks on the “Sign In” button, an Intent to “Sign In” is created using the GoogleApiClient. When it returns, the onActivityResult method of the current activity is called and the app obtains the GoogleSignInResult object. If the sign-in was successful, the app proceeds to obtain the name and email of the user from the GoogleSignInResult object and pass it to the intent created to start the next activity and subsequently start the Profile Activity

Profile - This screen is split into two fragments – “ProfileFragment” and “MapFragment”. The first section displays the data obtained from the Login activity. The second section displays the nearest stores to the user’s location and gives user the option to start scanning. Fragments have been used to create this layout.

ProfileFragment – This makes use of a ListView along with an ArrayAdapter to display the information.

MapFragment – This makes use of the Google Maps Android API v2 to obtain the GoogleMap object and display it on screen. Once the GoogleMap object is ready, the activity uses a LocationRequest object and a ResultCallback object to check if GPS is enabled on the user’s phone using a callback object. A Toast and an AlertDialog with an ‘ignore’ button is used in case it’s unavailable or switched off respectively. If it’s switched on, the activity uses Marshmallow’s new runtime permission model, to check if the app has permission to obtain the user’s location. If not, it asks for permission, results for which are delivered in a callback fashion as well. After user’s permission is obtained, a query is built using the user’s location co-

ordinates and a request is made to Google's Places API in an Async Task. For the sake of code brevity and modularity, the Async Tasks pertaining to maps have been moved to the BackgroundTasks class. The Async Task – NearbyStoresTask makes queries to the Google's Places API and collects the JSON data into a single String and passes it to another Async Task – MarkStoresTask. MarkStoresTask parses the JSON to obtain a list of store objects. The postExecute method notifies the MapFragment view when it's done and passes it the list of store objects. The MapFragmentView then reads the list one store at a time, creates a MarkerOptions object for it and proceeds to draw it on the GoogleMap object. Finally using the CameraPosition object, the map zooms into the user's current location. The user has to press the "Scan" button to proceed to the next activity

Scan – This screen uses the TableLayout which is dynamically populated. As and when the user scans a particular item it appears in the TableLayout. A user can simply long press the item to remove it from their list. When the user clicks on "Scan" button, the activity uses an intent to call the ZXing library which open's the user's camera, scans the barcode and returns the barcode ID to the activity. Once the activity receives the barcode ID, it queries the database to obtain the product's name and product's price and populate them in the TableLayout view. When the user clicks on "Proceed" button, the total amount and the entire list of products are passed as a Serializable object to the next activity.

Payment – This screen receives both the total amount and the Serializable bundle (list of products scanned). However, it displays only the amount and three payment options. Whichever option the user selects, the next screen is launched as a Child Activity. Once the child activity finishes its function, the onActivityResults method is called, which passes the Serializable bundle of list of products to the final activity – Confirmation.

Confirmation - This activity uses an Async task to parse the Serializable bundle of list of products and store them in the database. In postExecute method of this Async task, the user is displayed the transaction ID, the receipt and a button on click of which the user can share the receipt.

Action Bar - All the activities share a common action bar, which allows them to navigate to the current activity's parent activity as defined in the app's manifest file

Database Operations - As soon as the app starts, an async task is launched to create all the required tables and store the required data. Four tables – Barcode, Product, Order and TransactionEntry are created using DatabaseHelper class that extends SQLiteOpenHelper. This class also contains method to retrieve data and insert data into the tables. To facilitate this, model class for each table has been created.

Learning outcomes:

Throughout the development lifecycle of the app, vast knowledge was acquired many of which were technical skills and key skills such as standard Android/Java OO design principles and fragments, intents, cursors and activities to name a few . The importance of the versatile UI, was recognized. Throughout the project's lifecycle the team tested many different user interfaces, noticing along the way how small changes can make a big difference to making an app intuitive to a user. Different design ideologies

implemented all consisted of testing different color palettes, page layouts and overall app structure. Last, but by no means least was learning how to work in a small group. For a couple of students this was their first time working in group in a software development environment. For all involved key development methodologies, such as agile and XP (extreme programming) were implemented and studied. Learning how to collaborate and integrate personal work (code) were big learning points also. Lastly, 'soft skills' such as communication and interpersonal skills were extensively developed through constant teamwork and individual bi-weekly presentations of work done.

Future Development

Google Sign In

- The Google SignIn API requires a key. In development mode, the key is dependent on the developer's debug keystore found in the Android installation folder. So, the sign in can be achieved only when the app is run using Android Studio from one computer. However, when the app is released in production mode and uploaded to any marketplace, a production key can be obtained, post which the apk file can be installed on a phone and the sign in feature will work fine.

Server

- Currently, all the app data is stored on the phone. In a real-world scenario, user, product and order history would be stored on the server database. A prototype server application was developed using Tomcat, Hibernate and Postgresql and deployed on a laptop. However, the UCD Wi-Fi firewalls restricted requests from a mobile device to the laptop. In a real-world scenario, the server would be deployed on a machine that was publicly accessible and exposed to the app using a RESTful API.

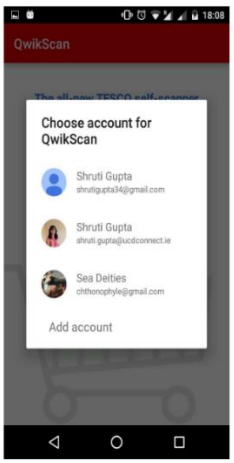
Payment Methods

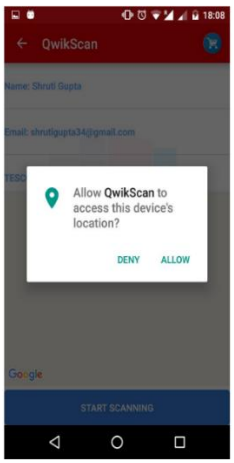
- As of now, the two methods of wireless payment that have been proposed are unfortunately not supported in Ireland. Although, they are available in different countries such as North America, Canada, japan, Spain etc. Though these technologies are not available right now, they are in the works. When looking through the careers connect website provided by UCD to aid in finding internship opportunities for us, it was noted that there was a listing for part time Android pay testers to test AIB and KBC bank cards, so obviously, this technology is at least being considered.

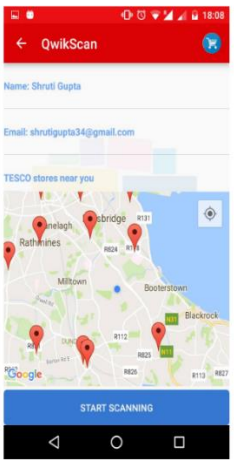
Conclusion:

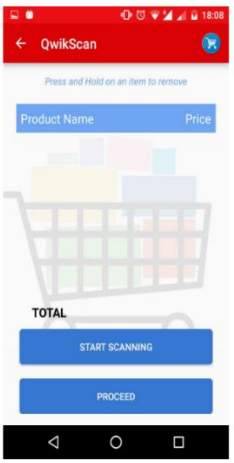
QwikScan unlike its competitors combines multiple payment options ensuring backward compatibility to give more options for its users. It is also loosely designed to be customized for any enterprise and finds nearest shops of the enterprise for the user. Due to technical and time constraints, features like Google Sign-In, Server hosting and end-to-end payment gateway are work-in-progress. QwikScan 2.0 can incorporate these. The app meets the project requirements with its versatile use of UI elements like Toasts, Lists, ActionBar, Fragments, use of camera and gps sensors, use of sqllite for storing data locally, use of Google Places API and calling another application to share data.

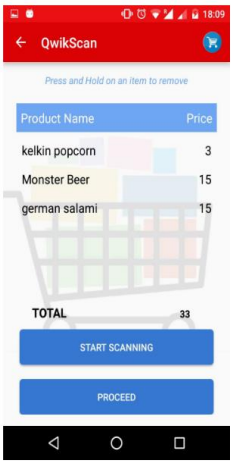
Appendix:

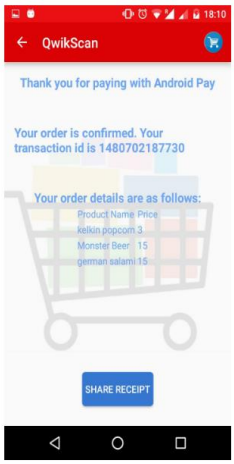












Barcodes

(Dummy data to be used with app –Please print in colour before using)

References

1. https://developer.android.com
2. https://github.com/zxing/zxing
3. https://developers.google.com/places/android-api/
4. https://developer.paypal.com/docs/accept-payments/express-checkout/ec-vzero/get-started/
5. https://support.google.com/androidpay/merchant/answer/6345242?hl=en