

Smart Waiter Detailed Design

Meraj Patel #1137491
Pavneet Jauhal #1149311
Shan Perera #1150394

January 11, 2016

Contents

1	Introduction	3
1.1	Purpose	3
1.2	Description	3
1.3	Scope	3
2	Overview	3
2.1	Revision History	3
3	Development Details	3
3.1	Language of Implementation	3
3.2	Supporting Frameworks/APIs	4
4	Implementation Components	4
4.1	Camera Function	4
4.2	Account Function	4
4.3	Ordering Functions	5
4.3.1	Parsing Function	5
4.3.2	MenuCategories functions	6
4.3.3	MenuItems functions	6
4.3.4	Users functions	6
4.4	Couchbase Functions	7
4.4.1	Creating local database	7
4.4.2	Creating local database	7
4.4.3	Replications	8
4.4.4	Sync Gateway	8
4.5	Error Handling	9
4.6	Communication Protocol	10
5	Database Structure	10
5.1	Implementation detail	10
5.2	JSON data format	11
6	User Interface Design	13
6.1	User Interface Design Overview	13
6.1.1	Menu Categories	13
6.1.2	Menu Items	14
6.1.3	Confirm Order	15

6.2	User Interface Navigation Flow	15
6.3	UI Description and Use Cases	16
6.3.1	Sign In Page	16
6.3.2	Barcode Scan Page	17
6.3.3	Menu Categories Page	17
6.3.4	Category Items Page	17
6.3.5	Customize Item Page	18
6.3.6	Cart Page	18
6.3.7	Payment Page	18

List of Figures

1	Main menu UI screen	13
2	Entree Category UI screen	14
3	Order Confirmation UI screen	15

List of Tables

1	Revision History Table	3
---	----------------------------------	---

1 Introduction

1.1 Purpose

The purpose of Smart-Waiter aims to provide a solution that will allow users to order and pay through a mobile application at restaurants.

1.2 Description

This opportunity arose from the lack of a universal application in the market that allows users to walk into a restaurant, scan a code to view the menu, and proceed to order and pay through the use of a singular application. Android users will be able to walk into any restaurant that offers our solution, and have the ability to use these services.

1.3 Scope

The scope of Smart-Waiter will be limited to providing the user with the following features: viewing the restaurant's menu, creating the user's order, placing the order and paying for their order.

2 Overview

2.1 Revision History

Date	Comments
January 11, 2016	First Draft.

Table 1: Revision History Table

3 Development Details

3.1 Language of Implementation

The source code of the Android Smart-Waiter application will be written in Java and XML, as well as the API libraries used in development.

3.2 Supporting Frameworks/APIs

CouchbaseLite (Database)

ZXing Embedded (QR Code Scanner)

Stripe (Credit Card)

4 Implementation Components

4.1 Camera Function

In this section we will detail the function that relates to the camera aspect of Smart-Waiter. There is one main function necessary for QR code scanning: `onActivityResult`. `onActivityResult` is called as soon as a QR code is captured by the camera. The function is passed three variables, a `requestCode` (int), a `resultCode` (int), an `intent` (Intent). Using these three variables, the function calls the ZXing function `parseActivityResult` using the three aforementioned variables as parameters. The `parseActivityResult` returns the contents of the QR code.

```
function onActivityResult (requestCode, resultCode, intent)
    IntentResult contents = parseActivityResult (requestCode,
resultCode, intent)
    if(contents is not null)
        String QRContents = contents.getContents()
        Call populate menu functions using QRContents as a
parameter
```

4.2 Account Function

In this section we will detail the key function that relates to account transactions and credit card charges. There is one key method related to credit card transactions: `chargeParams`, it involves using the Stripe class. As described in the System Architecture, given an instance of the Card class, and using Smart-Waiter's Stripe API key, a token is created and sent to the Stripe servers. The Stripe servers return a token that can be used to charge the user's credit card. The token is then used as a parameter of `chargeParams`, along with the information related to the credit card charge, like amount and the type of currency.

```

Stripe = new Stripe (Smart-Waiter's Stripe API key)
createToken(card, tokenCallback())

token = request the Stripe token from Stripe server

try{
    chargeParams(amount, integer amount in cents)
    chargeParams(currency, string value of type of currency)
    chargeParams(source, token)
    chargeParams(description, string value of description regarding
charges to card)

    charge = Charge.create(chargeParams)
}
catch (Exception e){
    Error handling
}

```

4.3 Ordering Functions

Below are function descriptions that are necessary for ordering in Smart-Waiter. As explained in System Architecture, there are three primary classes used to hold all vital information regarding menu information. These are: MenuCategories, MenuItems and User. These three classes are used to provide a user with a full menu of a particular restaurant and allow them to place an order.

4.3.1 Parsing Function

A parsing function was created to parse through received JSON responses to save information in appropriate classes. Stated below is pseudo code for this JSON parser. This function will parse through a JSON request. View if the key equals "categoryname". If so, save the value within MenuCategories class under variable, categoryName. Also, if the key equals "categorypic" save the value within MenuCategories class under variable, categoryPic.

```

function parse throws exception corrupt
    define categoryArrayList
    Save JSON response as list of hash map array
    for (each element in hash map list) {
        while (each array element with list) {
            if(key equals categoryname){
                Create MenuCategories object, save category name
            }
            if(key equals "categorypic"){
                use MenuCategories object, save category name
            }
        }
        add created MenuCategories object to categoryArrayList
    }

```

4.3.2 MenuCategories functions

Primary functions are accessors and mutators

getCategory: Get category name

getPic: Get category picture

4.3.3 MenuItems functions

Primary functions are accessors and mutators

getItemName: Get item name

getItemDetail: Get item description

getItemPrice: Get item price

4.3.4 Users functions

Primary functions are accessors and mutators

requestCode: Used to parse the QR code

resultCode: Used to parse the QR code

intent: Gives the current intent, used to parse the QR code

4.4 Couchbase Functions

4.4.1 Creating local database

A top-level manager object is created when the application launches, which manages collection of couchbase instances. In the case of this application, there are two database instances, one that holds menus and another to hold orders. The Manager creates a directory in the filesystem (if one does not already exist) and stores the databases inside it. The file path is determined by the application context. Thus, if the application is ever deleted, all of the content will be deleted as well. In addition, creating the database at this point makes it readily available for functions, which might try to use it immediately following the applications launch.

```
public CouchBaseLite(MainActivity AndroidContext) throws IOException,  
CouchbaseLiteException {  
    AndroidContext.manager = new Manager;  
    AndroidContext.database = manager.getDatabase( "menus" );  
    print("Created Couch Base Lite database");  
}
```

4.4.2 Creating local database

The CouchbaseLite API is used to access the documents by barcode. Furthermore, the couchbase get API is used to extract all menu details. An example of extracting a menu from the database is given below.

```
public Document getRestaurantByBarcode(String barCode){  
    restaurantMenu = getDocument(barCode);  
    return restaurantMenu;  
}
```


4.4.3 Replications

Couchbase API will be used to replicate data from the cloud. For the scope of the application a replication of the whole database is used. However, in the future when there are thousands of restaurants, a filter can be used to pull specific menus from the cloud on demand. Both a continuous pull and push sync are setup so the application can push and pull at any time the application is running. In addition, a change listener is set on each pull to confirm the status.

```
public void startReplications() throws CouchbaseLiteException,  
MalformedURLException {  
    Replication push = database.createPushReplication(url); Replication pull =  
    database.createPullReplication(url); pull.setContinuous(true);  
    push.setContinuous(true);  
  
    pull.addChangeListener(ChangeListener() {  
        @Override  
        public void changed(ChangeEvent event) {  
            // will be called back when the pull replication status changes  
            if (getStatus() == REPLICATION_IDLE) {  
                print("The replication is complete");  
            } else {  
                print(" The replication Failed");  
            }  
        }  
    });  
}
```

4.4.4 Sync Gateway

The Sync Gateway requires some basic configuration to correctly synchronize mobile devices and the Couchbase server. The basic configuration is given below. Firstly, the sync gateway is pointed towards the Couchbase server node and the data bucket, which holds menus and orders. Furthermore, the Sync gateway is assigned a sync data bucket. This bucket is necessary because the sync gateway needs to store meta-data for accessed documents, such as revision etc. Since we want to decouple the sync-gateway from the Couchbase server and the restaurant menus and orders data. A new bucket is created for storing sync gateway metadata.

```

    "couchbaseevents": {
        "server": "http://127.0.0.1:8091",
        "bucket": "Smartwaiter_menus",
        "shadow": {
            "server": "http://127.0.0.1:8091",
            "bucket": "sync_data"
        },
        "bucket": "orders"
        "shadow": {
            "server": "http://127.0.0.1:8091",
            "bucket": "sync_data"
        }
    }
    "import_Docs": true,
    "sync": `
        function (doc) {
            channel (doc.channels);
        }
    `
}

```

4.5 Error Handling

As per the pseudo code, functions throw appropriate exception when an error occurs. There will actually be an error-handling module called in the catch statement, which will handle all of the error in the program. The design will be a simple mapping between error code and how the error will be handled. In some cases for instance, when scanning an invalid barcode, the error is handled by propagating the error to the UI. Similarly, error which result in unexpected behavior will be propagated up to the UI. However, other errors such as replication fail due to network connectivity could be ignored and logged in the application log dump for debugging. They will not be propagated up to the UI. This module will most likely be changed according to feedback from end users.

```

try {
    //example function
    populate_array(array_of_items);
} catch (IndexOutOfBoundsException e) {
    /* Here error handler class decides how to handle the error.
       Ignore or propagate error up to UI? */
    handle_error(e);
}

```

4.6 Communication Protocol

There is only one communication protocol used within server-client inside of our application. This is used to pull information from the couchbase server. However, the application simply calls Couchbase Lite API to pull data from the server. The details are implemented by couchbase Lite class itself. An example of HTTP get is given below to pull data from server using a Mac terminal.

```
curl -X GET 192.168.2.12:4984/couchbaseevents/1
```

5 Database Structure

5.1 Implementation detail

The couchbase server uses some basic rules to store data on the cloud. Here are some of the following:

- Data is stored as key-value pairs, where the key is the barcode and the value is the entire menu of the restaurant.
- Buckets are basically equivalent of a database.
- Each item in the database is referred to as a document.
- The database is schema-less, thusly there is no rigid structure

5.2 JSON data format

As mentioned above the database is schema less, meaning the JSON format and attributes can change from one restaurant to another. However below is an overview of the general JSON formatted restaurant menu.

```

{
  "Res_Name": "Habibiz",
  "category": [
    {
      "type": "Appetizers",
      "url": "https://blah.com/pic"
    },
    {
      "type": "Main Course",
      "url": "https://blah.com/pic"
    },
    {
      "type": "Dessert",
      "url": "https://blah.com/pic"
    }
  ],
  "Appetizers": [
    {
      "name": "Buffalo Chicken Dip",
      "price": "12.99",
      "details": "Buffalo wing sauce, cream cheese and ranch make a great party dip."
    },
  ],
  "Main Course": [
    {
      "name": "Minced Beef Mushroom",
      "price": "12.99",
      "details": "Ground beef with soup-cream of Mushrooms"
    },
    {
      "name": "Beef diced potatoes",
      "price": "10.99",
      "details": "Stewed ground beef with minced potatoes"
    }
  ],
  "Dessert": [
    {
      "name": "Plum Pudding Cake",
      "price": "8.99",
      "details": "This old-fashioned dessert has a sweet cake layer on top and a juicy plum pudding filling beneath!"
    },
    {
      "name": "Blackberry Coffee Cake",
      "price": "10.99",
      "details": "A quick coffee cake with blackberries folded in and a cinnamon crumb topping!"
    }
  ]
}

```

6 User Interface Design

6.1 User Interface Design Overview

Below are sample screen mock ups for the UI in place for Smart Waiter

6.1.1 Menu Categories

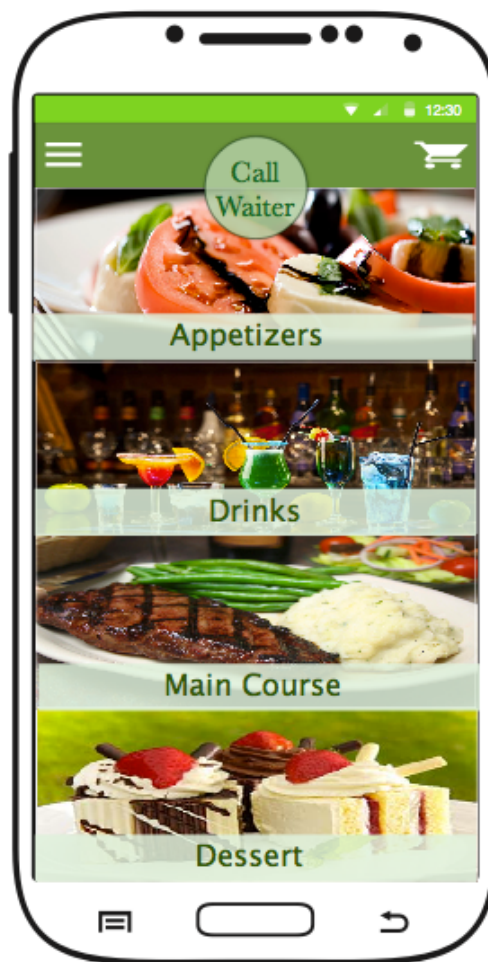


Figure 1: Main menu UI screen

6.1.2 Menu Items

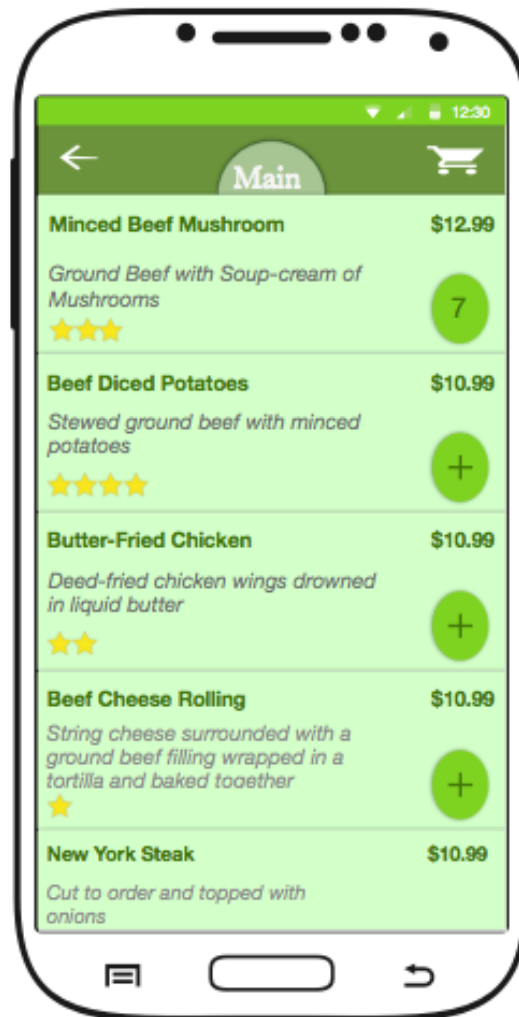


Figure 2: Entree Category UI screen

6.1.3 Confirm Order

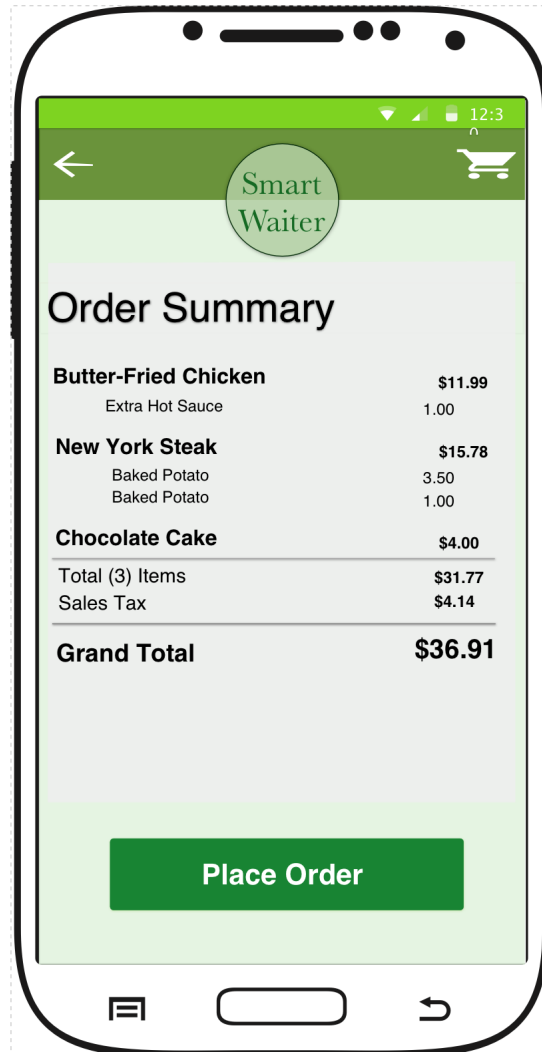
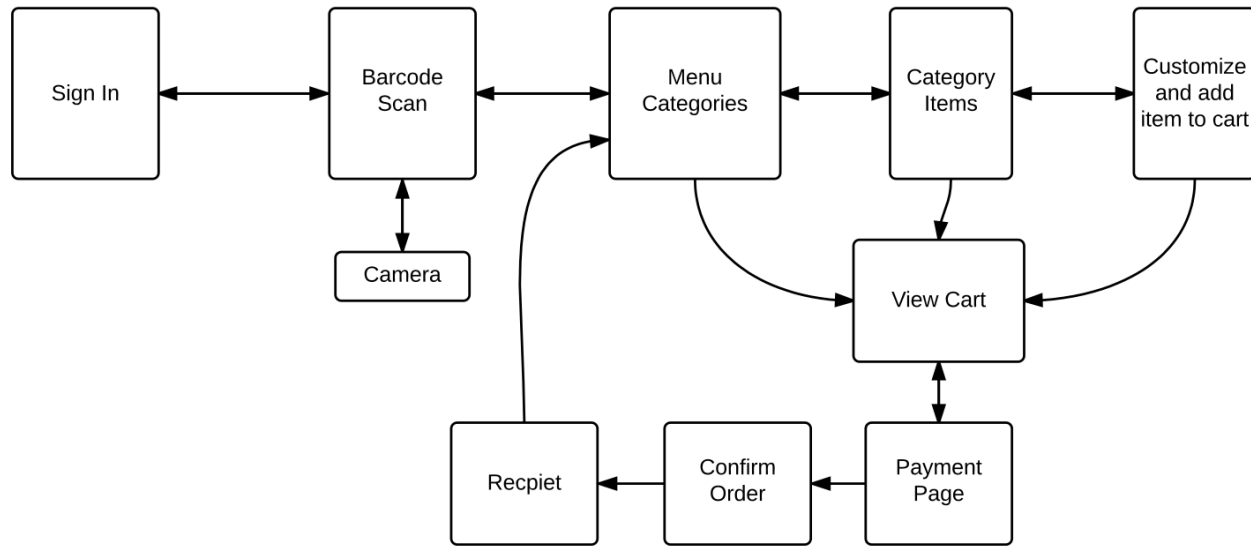


Figure 3: Order Confirmation UI screen

6.2 User Interface Navigation Flow

Presented is the over all UI navigation flow for Smart Waiter. Provided below are descriptions and use cases of each component.



6.3 UI Description and Use Cases

6.3.1 Sign In Page

When the app is launched the user is presented a screen promoting to sign in or skip sign in. The app will transition to one of two screens depending on the users decision. If the user decides to sign in, he will be redirected to a sign in page. Else, if he chooses to skip sign in, he will be redirected to barcode scanning page.

User Input	System Response
Enters correct user name and password	Application transitions to Barcode Scan page
Enters incorrect user name and password	Toaster displayed reading "incorrect login, please try again"
Enters incorrect user name	Toaster displayed reading "incorrect login, please try again"
Enters incorrect password	Toaster displayed reading "incorrect login, please try again"
Clicks "Skip Sign in"	Application transitions to Barcode Scan page
Clicks Back Button on phone	Application Quits

6.3.2 Barcode Scan Page

This page informs the user to locate the barcode presented on the table in front of him in the restaurant. After doing so, the user is asked to click Scan, to scan the bar code. Doing so, the app will transition to the camera page so the user can scan the code. Successfully scanning the barcode will transition to menu categories page.

User Input	System Response
Clicks Scan Barcode	Application transitions to camera so user can scan code. If successful, application will transition to menu page. Otherwise will return to scanning page and display a toaster reading, "please try again"
Clicks back button on phone	Application transitions to Sign in Page

6.3.3 Menu Categories Page

This page display menu categories in wide labels that cover the screen. Attached is the category name and a picture above. An example can be seen in 6.1.1. A user would simply click on a category to view its menu items. Doing so will transition the app to Category Items Page.

User Input	System Response
Clicks category	Application transitions Category Items
Clicks back button on phone	Application transitions to Barcode Scan

6.3.4 Category Items Page

This page display menu items in an android list view. Within each cell contains the item name, description and price. A use would simply tap on a cell to customize the item and add to cart. Please see 6.1.2 for an example.

User Input	System Response
Clicks Item	Application transitions to Customize Item
Clicks back button on phone	Application transitions to Menu Categories

6.3.5 Customize Item Page

This page allows customization of menu item. The UI consists of check boxes and options to indicate if certain toppings want to be added. The format of page is dependent on the type of item. At the bottom of the page, there will always be an option to add item to cart.

User Input	System Response
Ticks check boxes	None
Enters special instructions in input field	None
Clicks "Add to Cart"	Transitions to cart page and populate list with item
Clicks back button on phone	Application transitions to Menu items

6.3.6 Cart Page

This page displays all menu items a user requested. Below is an option to submit the order and proceed to payment page.

User Input	System Response
Clicks "Delete"	Deletes item from list
Clicks "Submit Order"	Transitions to payment page
Clicks back button on phone	Application transitions to previous page

6.3.7 Payment Page

This page allows users to input credit card information. This is the final page of the UI. After processing transaction, user will be redirected to bar code scan page.

User Input	System Response
Input valid credit card and clicks "Process"	Transitions into Confirm Order page
Input invalid credit card and clicks "Process"	Toaster displayed reading "invalid credit card"
Clicks back button on phone	Application transitions to previous Cart Page