

**Final Report: CHAI**

NET412

|  |  |
| --- | --- |
| Danny Yiu | 300675040 |
| Kenneth Sullivan | 300675040 |
| Marthe Nsaba | 300682252 |

Submitted to: Professor Hussein Fatmi

April 2, 2015

Table of Contents

Executive Summary 4

Mission Statement 4

Business Opportunities 4

Alternatives 4

Benefits 5

Financial Analysis 5

Product Description 6

Cost 7

Technical Planning 9

Logical Flowchart 9

Components 10

Server Setup 10

Web Service API Layer 10

Displaying order status: 10

Customer check-in: 10

Place order (checked-in customer): 11

Customer login/register/logout: 11

Customer update information (only performed when logged in): 11

QR scans: 11

Menu: 12

Menu editing (staff and admin only): 12

Store editing (admin only): 12

Internal Layer: Transaction Handling, Internal Logic, and Security 13

Logic Flowchart: 13

Interface Layer: 13

Security Layer: 13

PCI compliance: 13

Database Layer 14

Franchise Table: 14

Store Table: 15

Customer Table: 15

Customer App 15

Login/Register homepage: 15

Menu/Scan selection page: 15

Menu page: 15

Scan page: 16

Product page: 16

Payment page: 16

Order status page: 16

Profile page: 16

Store Web App 16

Admin Web App 16

Web service level: 16

Database level: 16

Development Roadmap 16

Risk Analysis 17

Patent infringement risks: 17

PCI compliance: 17

Development 17

Delays: 17

Vision misunderstanding: 18

Technical 18

Frontend app level: 18

Network level: 18

Conclusions, Recommendations and Suggestions 18

Recommendations and Suggestions 18

Amazon EC2 18

Security 18

Changes within the store 18

PCI Compliance 18

Conclusions 19

Foreword 20

Removed Features 20

Modified Features 20

Mobile platform: Cross platform 20

Mobile app design 22

Server platform: Conventional server 23

Payment system: PayPal API 23

Database 23

Web apps (cloud) 24

Bibliography 26

Glossary 26

# Executive Summary

CHAI is an order placing solution that caters to all types of cafes and coffee shops, such as Tim Horton’s and Starbucks, to expand their order capacity through a parallel transaction system complementing the existing. This report covers the business overview of CHAI and its full technical documentation.

This solution consists of two parts: the frontend mobile app and the backend cloud. The frontend app allow customers to check-in to a store via QR code1 scanning, purchase menu items, and view their order status as they wait for it to be completed. The backend cloud contains a database of order information, providing real-time transaction information to both the customer app and the store.

In essence, customers who purchase through the app can skip the usual lineup and go directly to the pickup counter. Whether there is one store or a thousand stores, the system’s functionality remains exactly the same. Not only can this increase store capacity, reducing lineup times will provide a better branding image and loyalty among the 90% of Canadians who have at least one cup of coffee every morning.

# Mission Statement

Our main goal and objective is to provide an effective and scalable complementary transaction system for coffee franchises using modern technologies. Our solution will increase store capacity and revolutionize the ordering system while being interoperable with existing systems.

# Business Overview

## Opportunities

Mobile apps have become an integral part of people’s lives. Consumers of smart devices spend 80% of their time on apps. Much like websites, apps are the new way for consumers to interact with businesses. Although the popularity of mobile apps is fairly recent, many businesses and industries are already recognizing and adopting apps into their frontend trademark; CIBC’s cheque scanning app last year, TD’s equivalent this year, Pizza Pizza’s 100000 app orders within six months, McDonald’s 2:1 ROI app just to name a few. Not only do apps provide a flexible and scalable utility, corporations are at risk of damaging their competitive edge and company image if they neglect the importance of apps for their businesses.

The specific business opportunities our app can provide can be summed up in two general directions: store capacity and company image. Store capacity will no longer be bottlenecked by the rate of transactions through the ordering lineup, which is the current situation. This problem can be especially obvious when augmented during peak hours, as lineups can extend all the way to the entrance of the store. Our app provides an alternative way to order without lining up, reducing the opportunity cost of potential customers being scared off by the lineups or leaving the line. Company image can also be improved by using modern transaction techniques, increasing intangible value such as customer loyalty. It also creates a competitive edge over other coffee shops without a similar transaction system, especially when customers become accustomed to it.

## Alternatives

An alternative to this is the use of vending machines. In fact, this is an existing method in Tim Horton’s, as they only recently began to use this approach in select stores. Users select and purchase their order on vending machines, acquire an order ticket, and head to the pickup counter. However, this approach can be costly and non-scalable. The hardware construction, installation and maintenance of each machine is the main cost, which can stack up quickly considering the amount of franchises Tim Horton’s own. Hardware infrastructure is also not GREEN, which may impact company image.

Another alternative can be for a company to create its own in-house app. An example is Starbucks’ experimental “Order-Ahead” app. This is a similar idea to our app, except developed internally. Their implementation reportedly increased their sales by over 13%. This is a perfect example of a successful app implementation into the coffee transaction system. Surveys suggest that customers find using an app to be a reliable method of making transactions (Ankeny, 2013):

***Chief Digital Officer Adam Brotman said 10% of all transactions in its U.S. stores are made with a phone. "Mobile devices have become an increasingly important part of the customer experience at Starbucks as the fastest and easiest way to pay in our stores," he said***.

The Starbucks app was well received, with an estimated 5.3 million downloads globally. It also increased speed of service, heightening customer satisfaction (Marko, 2013):

***CEO Howard Schultz said later in the call, the mobile app allows Starbucks to handle periods of peak customer demand much better.***

Our application has similar aspects to this but instead of every company or store developing their own application, we built a specialized solution. CHAI gives you worry-free, out-of-the-box implementation that is easy to use and maintained by our dedicated team.

## Benefits

Tangible benefits include higher store capacity, higher peak time revenue, low cost for expanding, ease of training and operation, high servicing and maintenance availability.

Intangible benefits include customer loyalty, more accurate ordering (no misspelled names or other human errors), and competitive edge over other coffee shops without similar technology.

## Financial Analysis

The return on investment (ROI) of our solution is difficult to calculate exactly due to its flexible nature, but we could estimate by using some example figures.

Since our solution has no technical limit on order capacity, it is only limited by the capacity of the kitchen. Assuming the kitchen is at 80% capacity when the cashier lineups are at 100% load, using the remaining 20% kitchen capacity wasted can potentially increase our total sales capacity by 25% (0.2/0.8=0.25). Our solution enables the use of this extra capacity since it works in parallel with the current ordering system. Our new maximum capacity will therefore be at 125%, or a maximum revenue increase of 25%.

Let’s consider a simple scenario: a large coffee franchise making an average of 13% increased revenue per month using our app installed in a single store. Assuming the store makes $2000 a day: that will equate to $200x0.13=$260 per day, or $7800 per month, per store of extra revenue.

The monthly costs for cloud hosting could scale to around $500 per month per store. The one time cost of developing the app and setting up is a maximum of $24000, which is only done during the first store.

Accounting for annual ROI for one store with setup costs paid off in the first year, it’ll be revenue ($7800) – monthly costs ($500) – asset installments ($24000/12) = $5300 per month for the first year, and $7800 - $500 = $7300 per month thereafter.

Of course, this is just a simple scenario for an estimate. The costs can widely vary depending on scale, as shown in the costs section below.

## Product Summary

The idea is simple: a customer at a participating coffee shop will be able to order menu items by scanning a QR code located in the store using their smartphone. Orders will appear in the store’s interface and can be marked as completed once fulfilled. This order status will be displayed to the customer’s app in real-time. Behind the app is a centralized cloud infrastructure handling individual store and client information, providing real-time order tracking capabilities to easily integrate into an existing order system. As such, any store with an Internet connection will be able to integrate this highly scalable solution.

|  |  |
| --- | --- |
| **Component** | **Description** |
| User App | Function:  The front-end interface of the solution for customers at the coffee shop.  Capabilities:   * QR code scanning * PCI compliant transaction through PayPal * Secure connection with cloud service through SSL * Android and iOS compatibility |
| Cloud Infrastructure | Function:  Centralized order tracking and information storage. Takes input from the user app and store interface, modifies central information, returns status to both the client and the store as order tracking.  Required Capabilities:   * PCI compliance of handling customer information * Interoperability or independent from existing ordering systems * Easy for client to manage the system, such as adding new stores, changing inventory details, or changing payment system * Low bandwidth, high traffic support (majority of bandwidth will be order information, which consists only of text) * Database to store order history such as inventory, current and past orders. Useful for sales data analysis. * Generation of QR codes |
| Support | Function:  Software setup, maintenance, and crisis response.  Required Capabilities:   * Instant response to emergencies * Help setup new stores when business expands * On-going support, may be either on-call basis or long term contract |

## Costs

|  |  |
| --- | --- |
| **Item** | **Options and fees estimate** |
| App and cloud setup | User app and cloud setup based on an in-house developed template. Will include full interface customization such as logo, color theme, images etc. to fit company image. Some basic functionality customization choices available.  *One-time fee: $4000*  *(1 programmer, $25/hr. for one month)* |
| Cloud Hosting | Amazon EC2 hosting.  *$50-$500+/month, flexible billing depending on resources used.*  PayPal2 credit card fees: *2.9% + $0.30 per transaction* |
| Support | Maintenance and upgrades of the system. Costs plans are flexible and scale with needs. |

# 

# Technical Documentation

## Overview

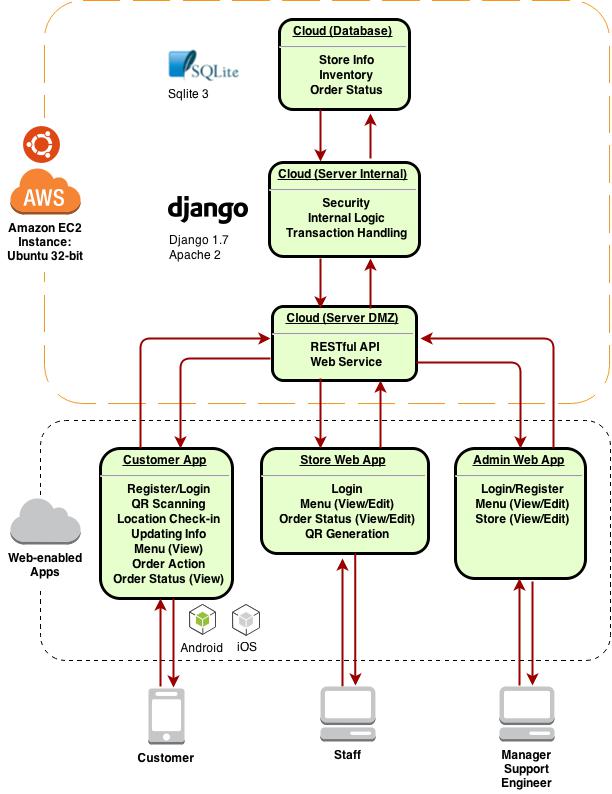


Figure : Logic flowchart

Our solution consists of two parts: cloud and app. Users, including customers and staff, use apps to communicate with each other through a cloud service. All orders are initiated by the customer (choosing and paying for an order) and terminated by the staff (completing the order). Admin users such as managers may add or edit stores through the web service as a separate entity from the staff and customer apps.

## Server First-time Setup

**Specifications:**

|  |  |
| --- | --- |
| Hosting Service | Amazon EC2 |
| Platform | 32bit Amazon Linux 2014.03 v1.0.7 running Python 2.7 |
| Web Framework | Apache 2, Django 1.7.1, sqlite33 |

Since our service backbone is entirely cloud-based, Amazon EC2 will handle all internal networking infrastructure. For our purpose, we will be using their East North America servers. Note that even though we are using Amazon EC2 for this documentation, our solution’s portability allows it to be installed under any web host with 32bit Ubuntu with the exact same configurations.

### Amazon EC2 Setup

Amazon Web Services (AWS) provide a feature called Amazon EC2, a collective toolbox for launching and maintaining instances, assigning IP address, managing security policies, and monitoring server status. As each instance is a virtual machine on a logical level, it can be logged in and accessed through a terminal interface using SSH like any other hosting service.

EC2 Website: <http://aws.amazon.com/ec2/>

**Minimum EC2 setup:**

* Single t1.micro or t2.mirco instance4
* One Elastic IP attached
* Inbound rules allow ports: SSH (22), HTTP (80), Websocket (1025), HTTPS (443)
* One keypair attached for SSH/SFTP access

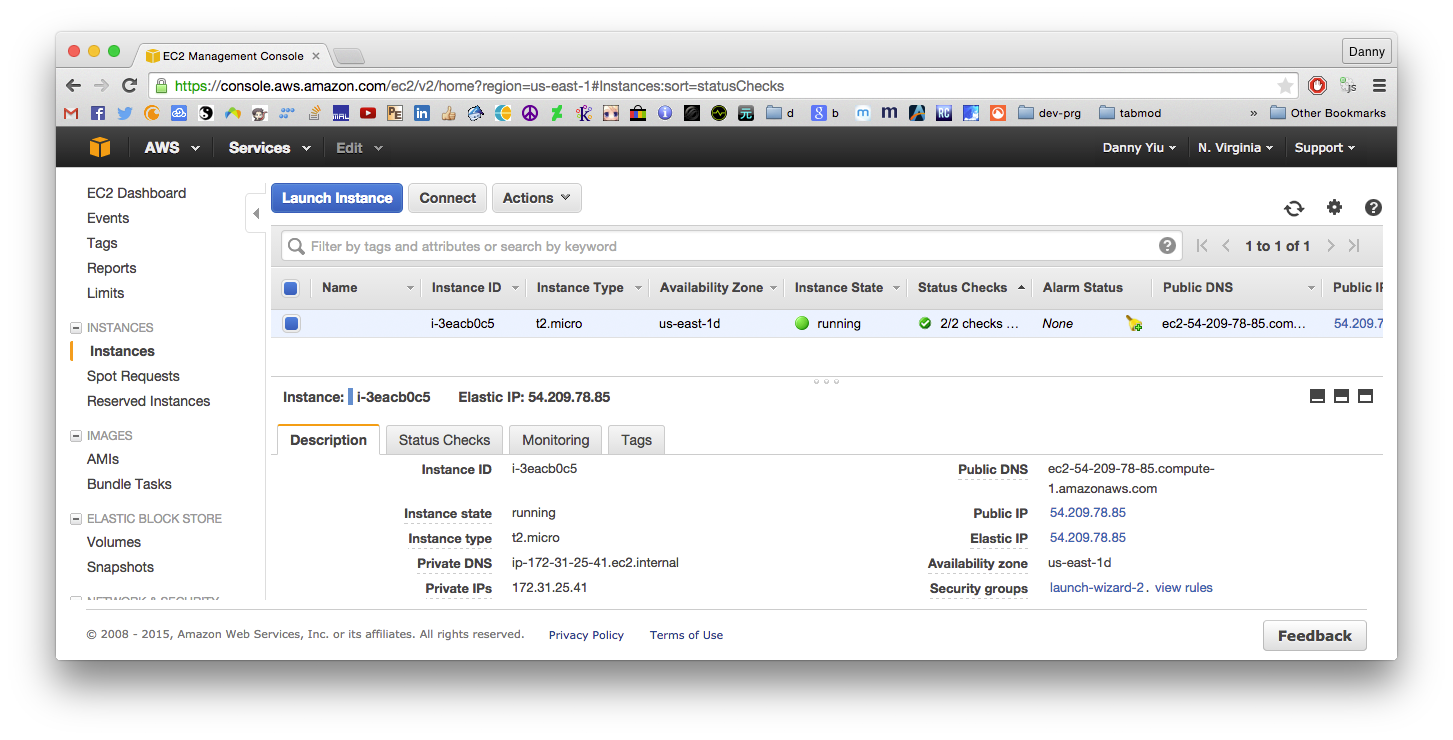


Figure : EC2 Instance

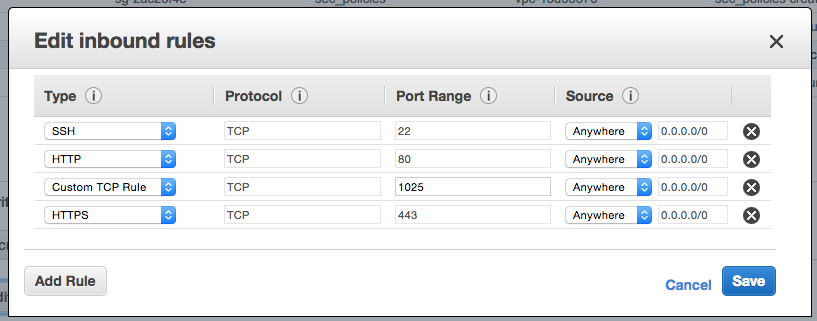


Figure : Inbound rules

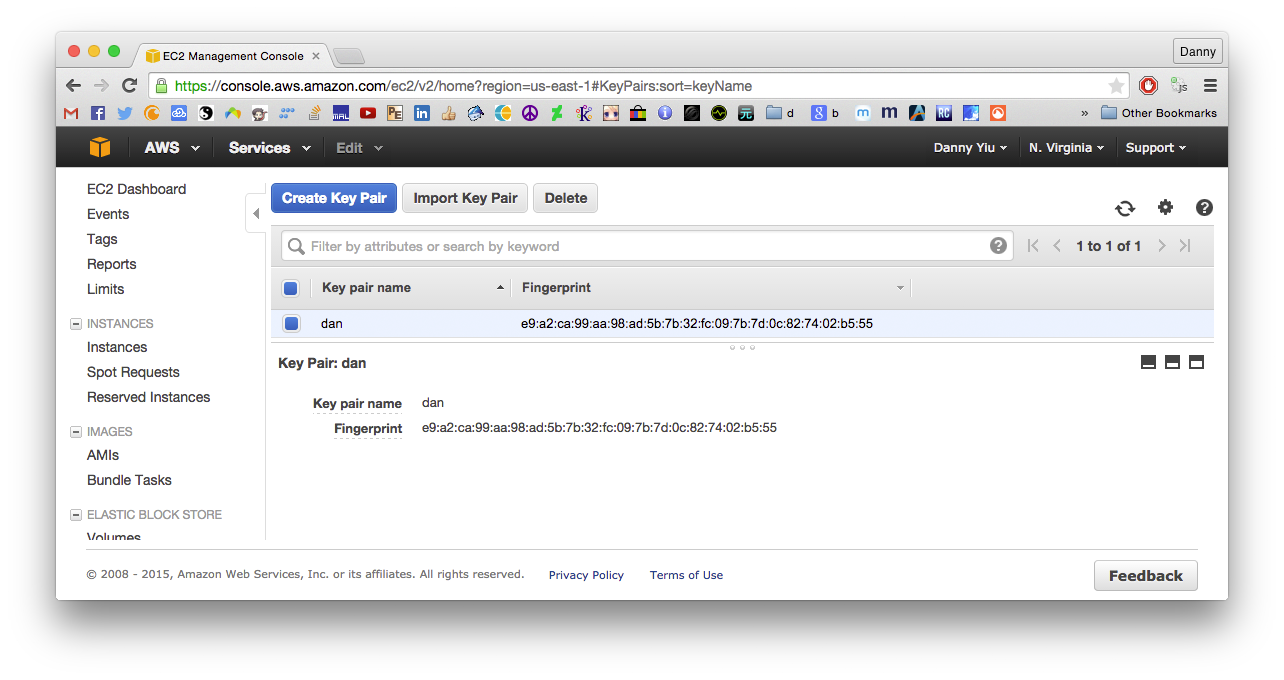


Figure : Keypair for SSH/SFTP. This is a .pem file used for authentication.

The Elastic IP is the public IP address of the instance, which can be used to update domain name DNS A records to attach a fully qualified domain name (FQDM). Alternatively, AWS Route 53 can also be used internally, which is Amazon’s DNS service. In this documentation, we will be using a domain name “www.chaiapp.tk”:

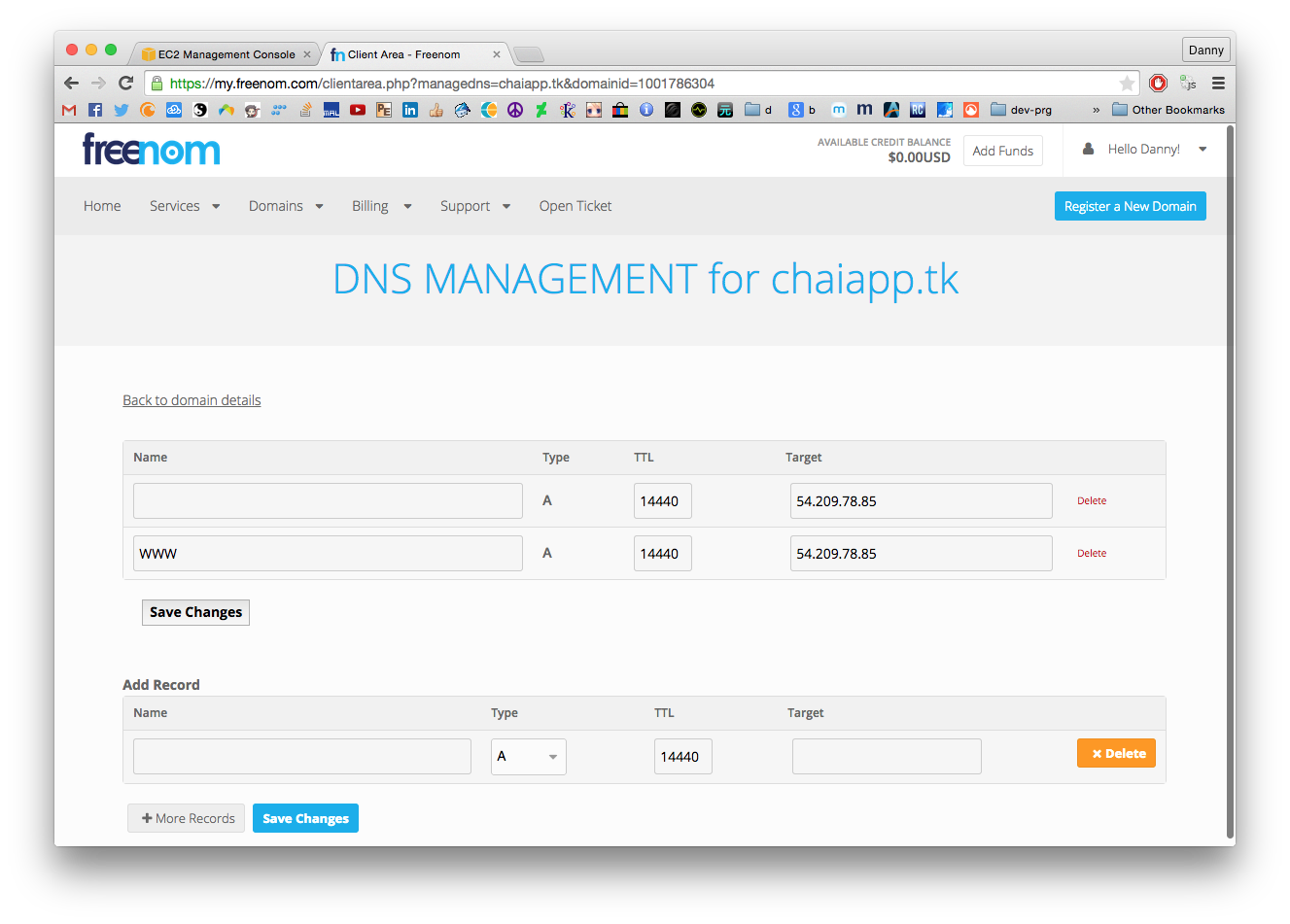


Figure : Updating DNS A records to add a domain name

### Backend installation

**Transferring files (SFTP):**

Once the instance is set up, we can configure the environment and install the backend. The first step is to copy the cloud backend package to the home folder. It is important to note that since setup scripts and configurations files have hardcoded paths, it is crucial the package must be copied to the home folder exactly (/home/ubuntu/).

Required:

* CHAI cloud package: <https://github.com/dannyyiu/410coffee/blob/master/release/chai_cloud_deploy.zip>
* SFTP client (i.e. Firezilla: <https://filezilla-project.org/>)
* SSH key pair attached to EC2 instance (\*.pem file generated above)
* SSH client

With the key pair file (\*.pem) generated earlier, use it to establish an SFTP connection with the server using an SFTP client. By default, EC2 Ubuntu instance should have a user “ubuntu”. Once connected, transfer “chai\_cloud\_deploy.zip” to path /home/ubuntu/. The compressed file can be removed at this point. There should be a director /home/ubuntu/chai\_cloud\_deploy/, this is the root folder for our backend Django project.

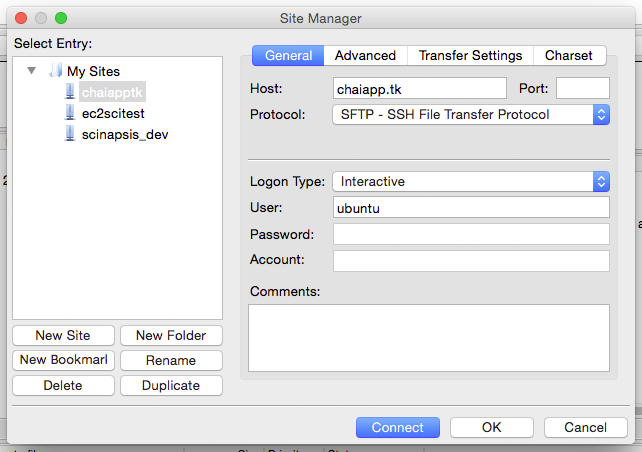


Figure : SFTP settings on Firezilla

**SSH:**

Requird:

* SSH client (SSH command in UNIX systems, or PuTTY for Windows)
* SSH key pair file attached to EC2 instance (\*.pem file)
* Username (by default, ‘ubuntu’ is generated by EC2)

Example ssh login in UNIX-based systems:

ssh -i dan.pem ubuntu@chaiapp.tk

**Server file structure:**

|  |  |
| --- | --- |
| **Path** | **Description** |
| /home/ubuntu/ | Home folder (~) |
| ~/chai\_cloud\_deploy/ | Project root folder |
| /etc/apache2/ | Apache configuration root |

**Server configuration:**

Configuration includes installing package dependencies, Python libraries, Apache module installation and activation, Apache configuration files, and directory ownership and permissions.

|  |  |  |
| --- | --- | --- |
| **Package Dependencies** | **Python libraries** | **Apache modules activation** |
| apt-get (only update required)  apache2  libapache2-mod-wsgi  python-setuptools  python-pip  python-dev  libffi-dev | django  httpie  twisted  pyopenssl (for 2.7)  twisted-websocket-4173-4  pytz  bcrypt | ssl |

**Permissions (all paths recursive)**:

|  |  |
| --- | --- |
| ~/chai\_cloud\_deploy/ | Everyone: Execute, write |
| ~/chai\_cloud\_deploy/static/ | Everyone: Write |
| ~/chai\_cloud\_deploy/media/ | Everyone: Write |
| /var/www/ | Everyone: Write |

Note: permissions set to everyone, but it mostly just for Apache. If this is a concern, recursively change ownership (chown) of all directories to the Apache user of your system.

**Apache configuration:**

|  |  |
| --- | --- |
| Location: | ~/chai\_app\_deploy/chai.conf |
| Copy to: | /etc/apache2/sites-enabled/chai.conf |

**Automation:**

For convenience, all of the above configurations are bundled in an installer script. To run the script, execute the following commands in SSH shell:

cd ~/chai\_app\_deploy/\_scripts/

sudo chmod a+x \_aws-setup

./\_aws-setup

If any errors occur, be sure to manually install missing packages.

**SSL:**

Unique SSL certificates must be obtained for every domain name. Once registered, three certificates must be obtained: Private key, public key, and an intermediate CA bundle certificate for the company you registered with. Public and private keys are generated during purchase (certificates for chaiapp.tk located in ~/chai\_cloud\_deploy/ssl/), while the CA bundle can be obtained from your SSL company website (RapidSSL was used for chaiapp.tk, CA bundle link: <https://knowledge.rapidssl.com/support/ssl-certificate-support/index?page=content&actp=CROSSLINK&id=AR1548>)

|  |  |
| --- | --- |
| Apache SSL configurations | /etc/apache2/sites-enabled/chai.config |
|  |  |
|  |  |

Once all three files are obtained, they must be placed in /etc/apache2/ssl-certs/ with the following filenames (if named differently, change accordingly in config file /etc/apache2/):

|  |  |
| --- | --- |
| Private key | rapidssl\_privatekey\_2015.key |
| Public key | rapidssl\_publickey\_2015.crt |
| Intermediate CA bundle | Rapidssl\_intermediateca\_2015.crt |

## Web Service API Layer

The Web Service API is the outer most layer, or the Demilitarized Zone (DMZ), of our cloud. For users to access our cloud, all requests by apps must be done through this layer. For some requests, hash authentication will be required for security reasons.

All Web Services are RESTful: HTTP requests sent to the webserver will respond with state data. For example, a POST request can contain customer name, order, payment information, and the server will respond with XML or JSON containing payment status. Below are some API methods:

### Displaying order status:

|  |  |
| --- | --- |
| Base URL | /order\_status |
| Description | Retreive tables containing order status |
| Parameters | |  |  | | --- | --- | | *id\_store* | Identification number of store. | | *table* | “current” to get a table of current orders.  “complete” to get a table of completed orders | | *len* | (optional) number of items to display. Default is 10. | | *start\_time* | (optional) display items from this time onwards. Format is *yyyymmddhhmm* | | *end\_time* | (optional) display items until this time. Format is *yyyymmddhhmm* | |
| Response | |  |  | | --- | --- | | *table* | JSON response containing table | |

### Customer check-in:

|  |  |
| --- | --- |
| Base URL | /check-in |
| Description | Verify customer’s presence in store using location-based services and store QR code. |
| Parameters | |  |  | | --- | --- | | *id\_store* | Store ID | | *store\_auth* | Authentication hash and public key using the store QR code. | | *id\_customer* | Customer ID | |
| Response | |  |  | | --- | --- | | *status* | JSON response containing “status”. | |

### Place order (checked-in customer):

|  |  |
| --- | --- |
| Base URL | /order |
| Description | Placing an order through the customer app. When the payment is complete, the order will show up on the store’s web app. |
| Parameters | |  |  | | --- | --- | | *id\_store* | Identification number of store. | | *auth* | Authentication hash and public key, a security feature to prevent unauthorized users. | | *id\_item* | List of item IDs to purchase | | *pay\_auth* | Hash for credit card payment | | *start\_time* | (optional) display items from this time onwards. Format is *yyyymmddhhmm* | | *end\_time* | (optional) display items until this time. Format is *yyyymmddhhmm* | |
| Response | |  |  | | --- | --- | | *status* | JSON response containing “status” | |

### Customer login/register/logout:

|  |  |
| --- | --- |
| Base URL | /account |
| Description | Register, log in, log out a user (log out also performed automatically) |
| Parameters | |  |  | | --- | --- | | *action* | “login”, “register”, or “logout” | | *user\_name* | Username | | *Auth (pass\_word)* | Password hash | | *email\_address* | (only for register) Email address | | *payment\_information* | (only for register) List containing payment info, contents are hashed. | | *user\_info* | (only for register) List containing user information | |
| Response | |  |  | | --- | --- | | *status* | JSON response containing “status” | |

### Customer update information (only performed when logged in):

|  |  |
| --- | --- |
| Base URL | /update\_info |
| Description | Update user information |
| Parameters | |  |  | | --- | --- | | *user* | Username | | *session* | Session information (hashed). | | *email* | (optional) Email address | | *user\_info* | List containing user information to update. | |
| Response | |  |  | | --- | --- | | *status* | JSON response containing “status” | |

### QR scans:

|  |  |
| --- | --- |
| Base URL | /qr |
| Description | Perform action based on QR code scanned. Actions can be dynamic, but only action status will be returned in the response. |
| Parameters | |  |  | | --- | --- | | *user* | Username | | *session* | Session information (hashed). | | *qr* | Link redirect, actions to perform, and/or information to retrieve from QR scan | |
| Response | |  |  | | --- | --- | | *status* | JSON response containing “status” | |

### Menu:

|  |  |
| --- | --- |
| Base URL | /menu |
| Description | Show in-store menu. Sold out items will be displayed too, but quantity marked as 0. |
| Parameters | |  |  | | --- | --- | | *id\_store* | Identification number of store | | *category* | (optional) display only the categories in this list. | | *name* | (optional) search inventory for item name(s) in this list | |
| Response | |  |  | | --- | --- | | *table* | JSON table containing menu items queried | |

### Menu editing (staff and admin only):

|  |  |
| --- | --- |
| Base URL | /menu\_edit |
| Description | Edit menu items by adding/deleting/editing inventory. Can only be used in admin and staff mode. |
| Parameters | |  |  | | --- | --- | | *action* | “add”, “delete”, “edit” can be used. Only one action per call. | | *id\_store* | Identification number of store | | *session* | Hashed session information. This determines logged in store/admin. | |
| Response | |  |  | | --- | --- | | *table* | JSON table containing edited rows | |

### Store editing (admin only):

|  |  |
| --- | --- |
| Base URL | /stores |
| Description | Add/Delete/Edit stores |
| Parameters | |  |  | | --- | --- | | *action* | “add”, “delete”, “edit” can be used. Only one action per call. | | *user* | Username | | *session* | Session information (admin session). | | *add* | List of column strings. Create a new row in the stores master table, and create a store table for the new store using the default template. Only “store\_name” is required, but other columns can be set using this list. | | *delete* | String representing store ID. This deletes the store from the master table as well as the store table. | | *edit* | List containing store ID and columns to edit. | |
| Response | |  |  | | --- | --- | | *status* | JSON response containing “status”. | |

## Internal Layer: Transaction Handling, Internal Logic, and Security

This layer lies between the web service layer and the database. Since it is behind the Web Server layer, it is isolated from all external access, making it secure for internal server processes and transactions. It functions as the security filter, transaction handler, and general process layer for all internal server scripts.

### Logic Flowchart:

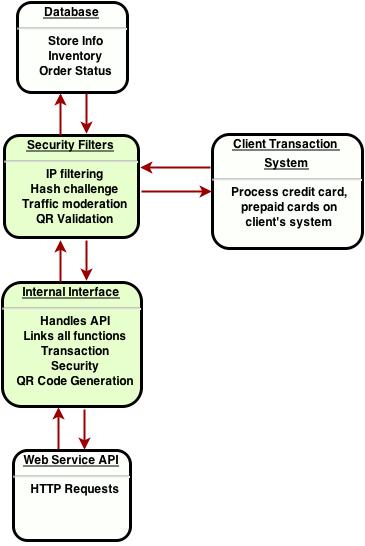


Figure 7: Logical flowchart of the cloud internal layer

### Interface Layer:

All internal functions are linked together by an interface layer. This layer then calls modules and functions to handle internal functions, such as transaction, security filters, QR code generation and validation, and translate API to database calls. The return values are then passed back to the Web Service layer.

### Security Layer:

Most interface requests to the database layer must pass through security filters. The main purpose of this layer is to detect and prevent misuse of the system. Some features include:

1. IP logging and filtering of flagged users
2. Hash challenge and response functions for authentication purposes
3. Validating QR code values to prevent forged QR codes
4. Traffic moderation functions to log statistics of traffic flow, both for marketing intelligence and flagging unusual activity.

### PCI compliance:

There is no one-size-fits-all solution to security. Instead, a layered approach must be used. To achieve PCI compliance, several layers will be used to protect customer information during transactions:

1. Passwords, session information, and credit card information will all be salt hashed
2. Payment information will not be stored on the mobile app
3. SSL will be used to encrypt interactions on the network traffic level.
4. Amazon EC2 servers provide secure physical and network infrastructure for server security.
5. Depending on the situation, a SMS validation service can also be provided for additional authorization security.

## Database Layer

The database contains all information used by the cloud service. There are few major tables that make up our database:

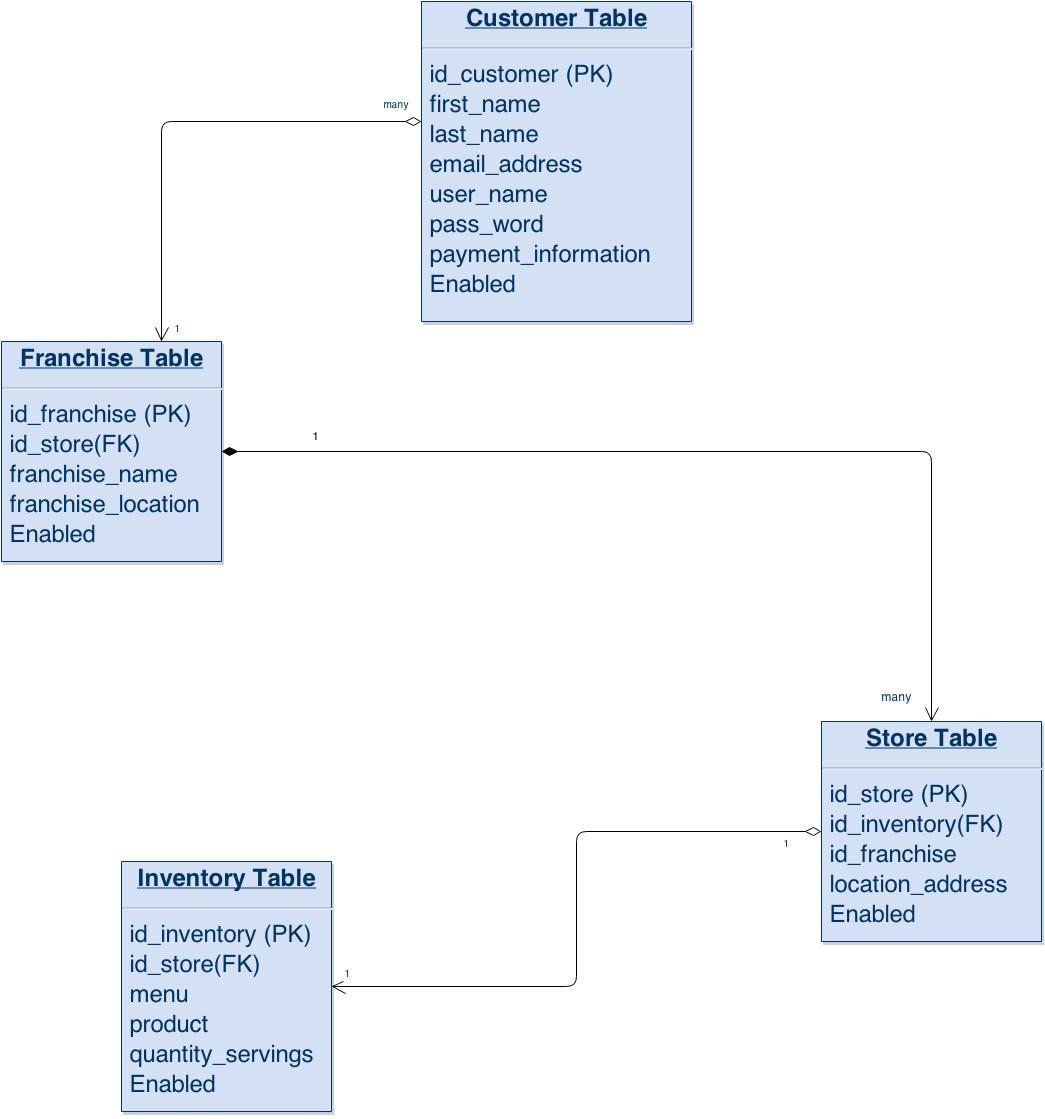


Figure 8: Database schema

### Franchise Table:

This is the highest level of the database hierarchy, containing store information such as location of store, store ID and which stores are active.

### Store Table:

This is a table used to store information about each individual store. Information can include inventory ID, menu and QR codes. It may also store order status.

### Customer Table:

This is a table used to store customer information. Special hash methods will be used to ensure PCI compliance of storing credit card information and passwords. Customer login and register functions will rely on this table.

Other tables, such as a separate table for each inventory, may be used to assist functionality. However, the above are the minimum requirements for our solution to function.

## Customer App



The customer app is a mobile app used to interact with the cloud service through API calls. An account can be created for each customer, but functionality and interface will the consistent.

The general pages for the app:

1. Login/Register homepage
2. Menu/Scan selection page
3. Menu page
4. Scan page
5. Product page
6. Payment page
7. Order status page
8. Profile page

### Login/Register homepage:

It contains login and password fields for login, a login button, a register button, and a guest button. Login success will lead to the Menu/Scan selection page, Register button will lead to a frame to enter user information for account creation. The guest button lets users log in as a guest.

### Menu/Scan selection page:

This page consists of two buttons: Menu and Scan. Menu will lead to the Menu page, Scan will lead to the Scan page. The Menu button will only be usable if the user is using the in-store Wi-Fi access point. This is to verify user’s presence in the store.

### Menu page:

All menu items for the store will be displayed here. The layout is flexible depending on client needs. The default is a table containing all menu items and available options for each item. Clicking any item will lead to the Product page.

### Scan page:

A QR code scanning plugin will be used for this page. It activates the camera, taking any picture of any QR code will lead to the Menu or Product page. Since all QR codes will be provided in-store, scanning QR codes verifies user’s presence in a store.

### Product page:

Product information, additional options available and price will be displayed. An “Add to cart” and “Checkout” button will be available. Cart can be displayed using a dropdown menu. The Checkout button will lead to the Payment page.

### Payment page:

Verifying order, final total calculation, payment options (credit card or prepaid) can be chosen. Only registered users may use saved card information. Guest users must enter credit card information every time.

### Order status page:

This lets users track their order. An order number will be used to identify the customer’s order, and a status of “Complete” will be displayed when the store is done preparing the order for pickup.

### Profile page:

Users may update their user information here. This can include their name, email, and payment information. No payment information will be saved on the mobile device for security reasons.

## Store Web App

The store web-enabled application is a webpage hosted in our cloud where store employees can login and use API to access the cloud. An account will be opened for every store, but the functions and interface will be consistent for all stores. Main functions include viewing and real-time orders, confirming order completion, and editing inventory status.

## Admin Web App

Like the Store Web App, this is also a webpage hosted in our cloud. Admin users such as franchise managers and support staff may login to this app to perform admin tasks such as monitoring store statistics, add/edit/remove stores, and add

### Web service level:

All cloud interaction must be done through the DMZ layer with specific API, each with specific permissions based on the user type (customers will have purchasing API, store staff will have inventory API etc). Access to the inner cloud is only limited to system administrators.

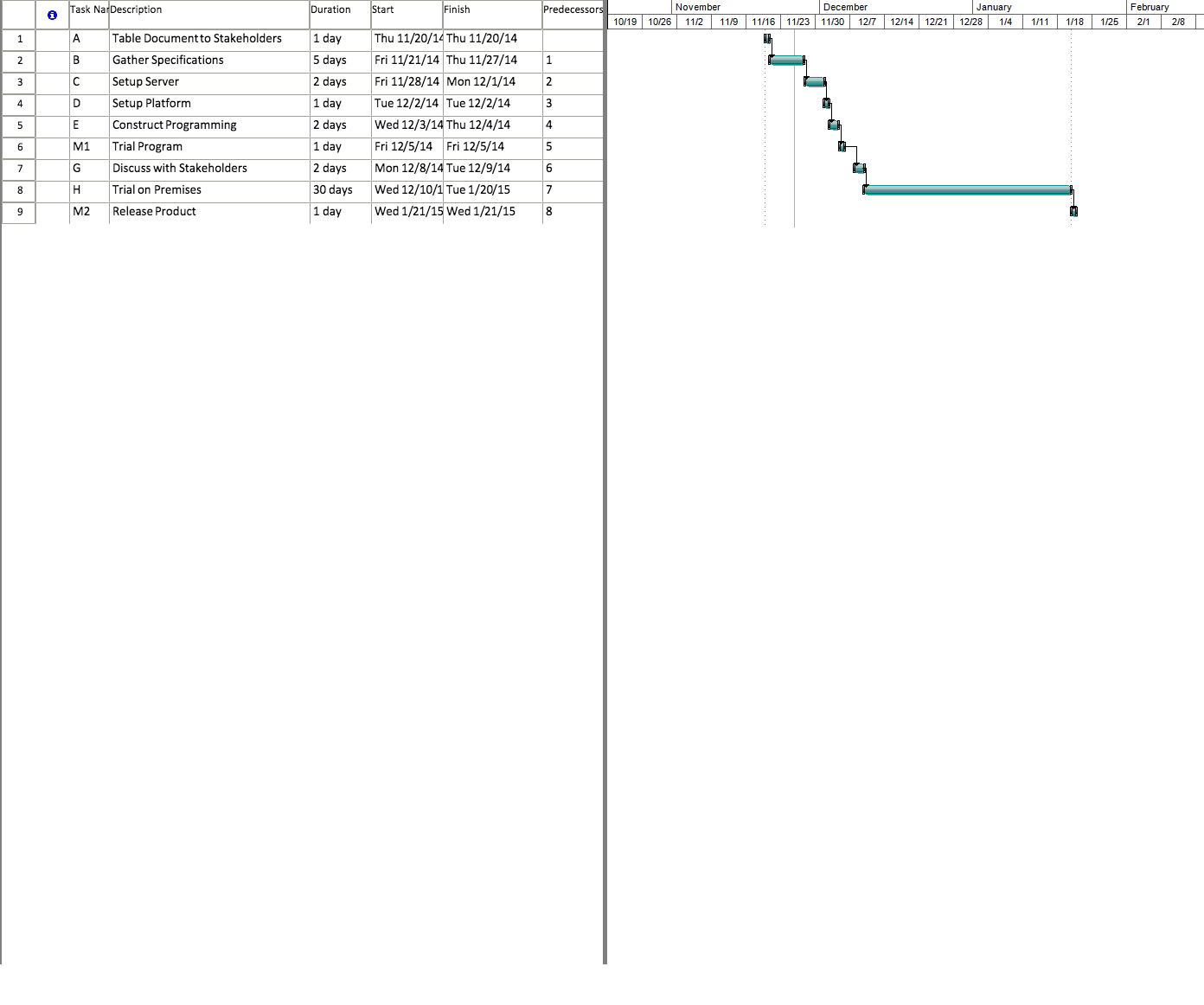
A security layer will log all network activity, monitor API calls and perform encryption. All connections from the DMZ to the inner cloud must pass through this layer.

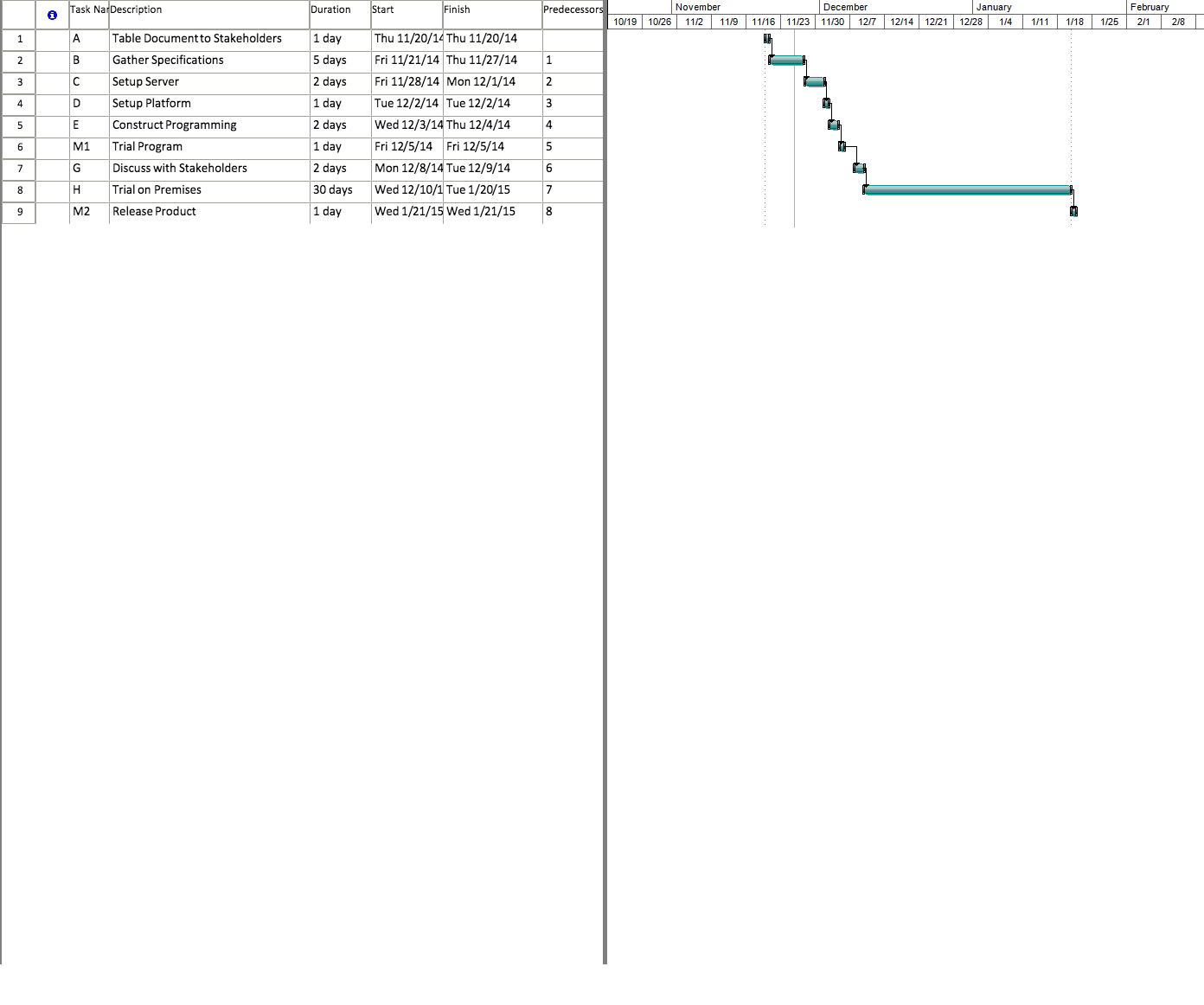
### Database level:

If the client decides to store credit card information with our cloud, all sensitive information will be hashed before storing.

# Development Roadmap

As our development is based off a template, most of the development time will be based on customization. Below is a timetable for our development:





# Risk Analysis

### Patent infringement risks:

Starbucks barcode app, along with Expedia Inc. and Capital One Financial Corp., sued by Maxim Inc. in 2012 for patent infringement of making mobile payments and securely transferring data and “cash equivalents” between devices (Wester, 2012). This specific patent does not threaten our app as barcodes are used only to transfer inventory and menu information, not transactions; secure data for transactions are processed just like any other web-based eCommerce method. However, this is a good example showing patent searching should be done thoroughly before the product design is finalized.

### PCI compliance:

Law suits can occur for companies disregarding PCI, or PCI DSS, compliance. Although some areas of PCI compliance are considered grey area, they do serve as general guidelines as to how sensitive information should be handled depending on the size of the business. Our cloud service is already compliant as we are using Amazon’s network and servers. Sensitive information is securely encrypted, sent over an encrypted network, and stored only if necessary. The details are discussed in the Technical risks below.

## Development

### Delays:

Our app follows a general template, which means development schedule for most cases are fast and require minimal developers.

### Vision misunderstanding:

Software solutions can often end up drastically far from the client’s original vision. While we will eliminate this by working closely with the client in all stages of the development, this risk is further reduced as our solution is a relatively light project.

## Technical

### Frontend app level:

Android file systems, regardless of rooted or not, can be directly accessed using Android Debug Bridge (adb), where files can easily transferred to a computer. Therefore, sensitive information such as credit card numbers will not be stored in the app. Furthermore, configuration will not be stored in separate files, but instead be embedded in the source code to be compiled. This eliminates clues external files can expose about the app.

### Network level:

All connections will be encrypted with SSL for protection against packet sniffing. Authentication information, credit card information and other sensitive data will also be salt hashed.

All connectivity for a particular store must be made within the store. This is enforced in two ways: Users who use the in-store Wi-Fi will all have the IP address of the store, automatically allowing menu access; Users who prefer to use their cellular data connection will have to scan a store-specific QR code to prove their physical presence in the store before menu access is permitted.

# Conclusions, Recommendations and Suggestions

## Recommendations and Suggestions

### Amazon EC2

Cloud hosting can be flexible depending on needs. Amazon EC2 hosting charges per usage hour, which is most suitable for our requirements for peak hours and off-hours balance. Amazon also has a strong reputation of secure servers for eCommerce, which can provide PCI compliance at least on the hardware and network layers. Designed for business, we can expect server performance even during heaviest workloads.

### Security

Customers may only order at the store they are physically in due to the security risk of remote attackers making bulk false orders. Location services and in-store barcodes may be used to implement this. Dedicated Wi-Fi or Ethernet may be used specifically for the store to access the server to isolate server access to stores only and minimize the risk of man-in-the-middle attacks.

### Changes within the store

Wi-Fi should be facilitated for customers who do not have phone data plans to access the cloud. This could be done with a small monthly fee depending on the location’s network providers.

A special lineup for the pickup counter for app users may be helpful both to eliminate order confusion and to let other customers be aware that the app is a new way to order.

### PCI Compliance

With more purchase volume, PCI Compliance becomes stricter. To prevent legal issues, it is important for a company to analyze which level of PCI compliance must be met. These are guidelines for both software and hardware considerations for secure handling of customer information, especially payment information.

## Conclusions

Mobile apps are not only a scalable tool for more efficient retail sales, they also bring a tremendous competitive edge to businesses who join the trend. Previous examples show high customer adaptation rate to these systems as they are already familiar with mobile apps.

Our solution is flexible depending on needs. Whether it is a simple setup for a single store, or a wide implementation of an entire franchise, we can tailor our solution to meet different business requirements and budgets.

The deployment of this solution is very lightweight, requiring only order status handling. With so little moving parts, this makes our system easily solid and secure. Depending on business requirements, we also consider PCI compliance into our design to ensure secure handling of consumer information.

Future expansion of stores is as easy as adding a new store in Admin Web App. The only additional cost is the hosting bandwidth and storage required, which is economical as it scales depending on need with Amazon EC2.

Future improvements of this solution could include server redundancy to improve uptime for high volume stores. With all user orders stored in the database, business intelligence applications can also be built with the data collected for statistical analysis.

# Foreword

We will be developing an app that caters to all types of cafes and coffee shops, such as Tim Horton’s and Starbucks, to expand their order capacity through a new transaction system complementing the existing.

This app consists of two parts: the frontend mobile app and the backend cloud. The frontend app allow customers to scan and/or choose menu items of the current store they are in, purchase them, and view their order status as they wait for it to be completed. The backend cloud contains a database of order information, providing real-time transaction information to both the customer app and the store.

The idea is simple; a customer at a participating coffee shop will be able to order menu items by scanning a barcode using their smartphone. Behind the app is a centralized cloud infrastructure handling individual store and client information, providing real-time order tracking capabilities to easily integrate into an existing order system. As such, any store with an Internet connection will be able to integrate this highly scalable solution.

This report addresses **technical scope changes only**. The business functions and technical structure remains the same since the original proposal.

Up to date project files are in our repository: <https://github.com/dannyyiu/410coffee>

# Removed Features

Due to time constraints, below are some features that are taken out of the project.

* SMS verification
* VPC for client payment (Changed to PayPal API)
* Amazon RDS, MySQL (Changed to sqlite, tested to be sufficient)

# Modified Features

## Mobile platform: Cross platform

Instead of targeting native Android only, we decided to make our app available for both Android and iOS. This is done using Monaca, a PhoneGap-based cloud compiler platform using HTML5 to develop cross-platform mobile apps. Furthermore, since HTML5 is used, we can fully integrate our cloud service with the mobile app with more interoperability.

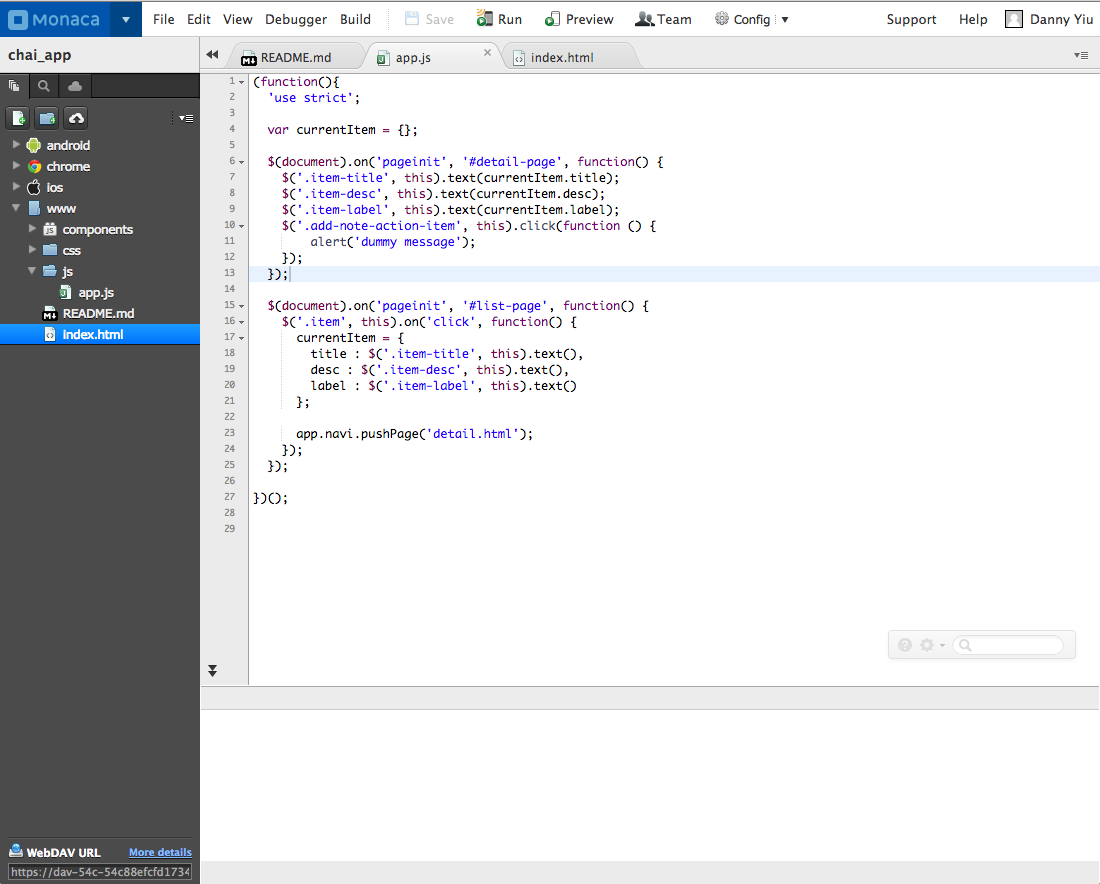
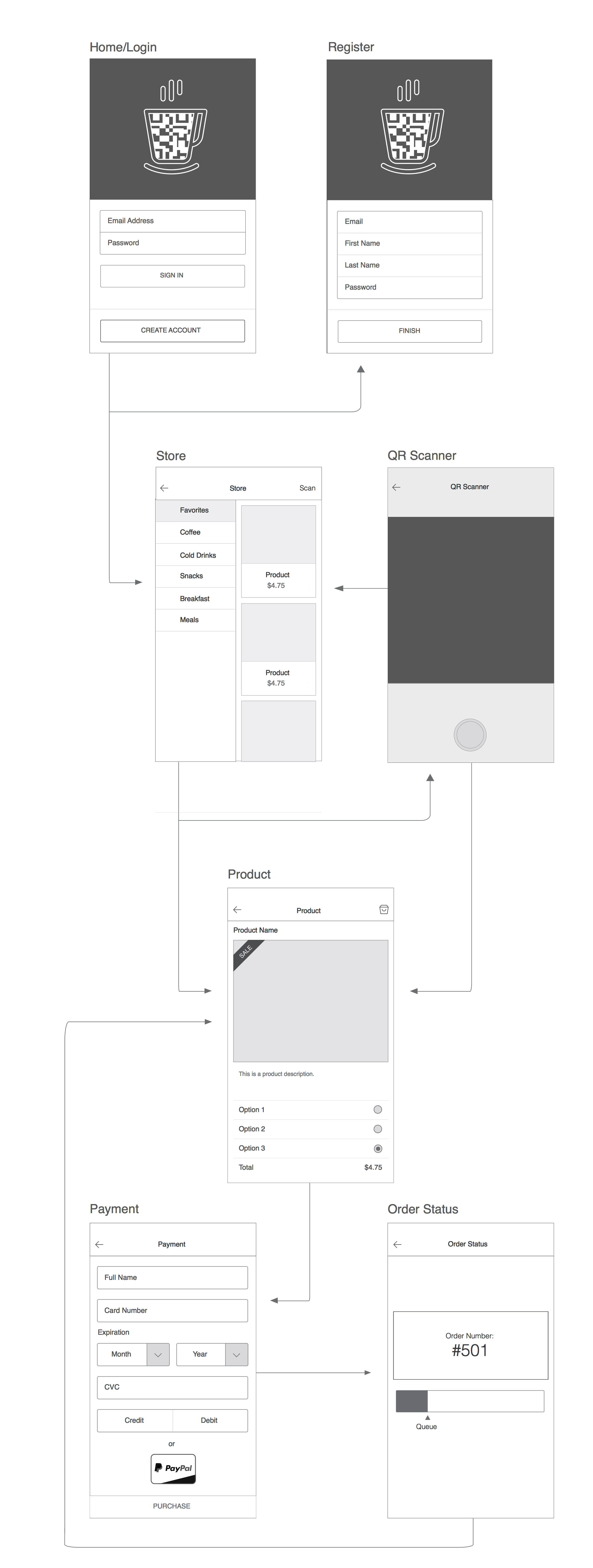


Figure 9: Monaca interface

## Mobile app design



Most app pages were kept the same, but now the menu and scan views are merged together in the Store view.

Basic flow:

1. User login or register from the homepage
2. User arrives at the Store page. This page will show a menu if using in-store wifi; otherwise, user is prompted to scan the store QR code to verify presence in store. A scan button remains in the menu bar regardless, allowing the user to scan other QR codes such as special in-store menu items.
3. QR scanner will arrive at the Store page if Store QR is scanned; will arrive at Product page if Product QR is scanned.
4. Product page will show product details, selectable options, and local cart functionality (add to cart, show cart, update cart, and checkout).
5. Payment page will redirect user to PayPal, which redirects back once transaction is complete.
6. Once transaction is complete, Order Status page will be shown, displaying order details in real-time.

All requests to our cloud will be done through our web service API, encrypted with SSL.

Figure 10: Wireframe of the mobile app logic flow

## Server platform: Conventional server

Our original plan to use Amazon Elastic Beanstalk proved to be clumsy for Rapid Development. Issues such as permission clashes when using local git- which is our project control handler system, environment variables not automatically flushing, and general slow updating caused us to reconsider. Currently, our project is deployed using the conventional way of SSH/SFTP for server communication and Apache 2 for hosting.

## Payment system: PayPal API

Instead of using Amazon VPC to connect to client’s payment servers, we decided to use PayPal’s API both to save time and to enhance security. Payment screens in the mobile app will be redirected to PayPal’s payment iframe, where all interactions are isolated to PayPal’s domain. This ensures no data is stored locally, and security is handed to PayPal’s sophisticated systems. Since PayPal API was designed for web, it can be integrated with our HTML5 mobile app without extra work.

## Database

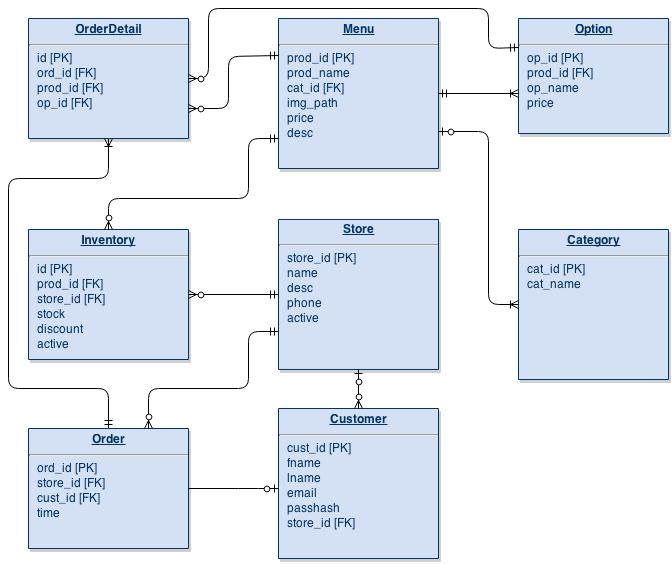


Figure 11: Database refactored for better organization and performance

Globally static tables: Menu (with Option and Category), Store, Customer.

Store-specific tables: Inventory, Order

Order-specific tables: OrderDetail

Since our ordering queries are relatively simple, we decided to use sqlite (specifically sqlite 3) to simplify deployment. Sqlite is tested to support up to 50k queries per second with no problems. That equates to 50 queries per second even at 1000 stores, which is well within our requirements even without optimization.

## Web apps (cloud)

Below is an updated flowchart of our cloud architecture.

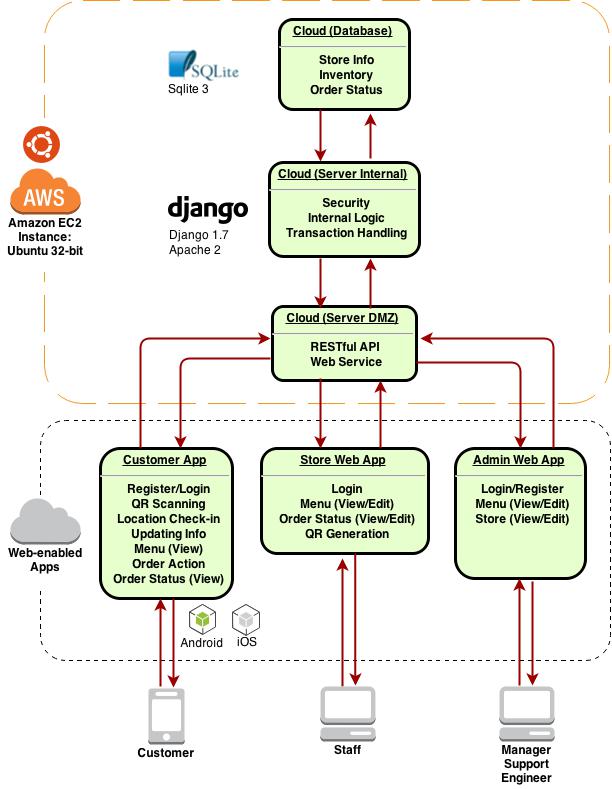


Figure 12: New logic flow

Although most interactions with our cloud are done through a web service API, user interfaces are required for the Store and the Franchise.

The Store web app is the interface in which the store operators use to view and complete new orders. Customer orders through the mobile app will be displayed in real-time on this interface, where store employees can mark as complete once they are finished with the order.

Real-time is achieved by websockets using a separate port for listening (1025). We accomplished this using Python’s Twisted and pyopenssl libraries for backend and Javascript AJAX calls for frontend. (Current sample deployed at: <http://chaiapp.tk>)

The Franchise web app is the interface in which franchise managers can view, add, modify and delete stores, menu items, and inventories. This can also be useful for viewing statistics. The Franchise web app serves as the administrator page for the entire franchise.

# Bibliography

Git. (n.d.). Git --local-branching-on-the-cheap. Retrieved from http://git-scm.com/

GitHub Inc. (2015). GitHub. Retrieved from <https://github.com/>

GitHub, Inc. (2015). *Django*. Retrieved from <https://github.com/django/django>

Python, Central. (2011). *Python SQLite Tutorial*. Retrieved from <http://www.pythoncentral.io/series/python-sqlite-database-tutorial/>

Asial Corporation. (2014). *Monaco.* Retrieved from<https://monaca.io/>

# Glossary

API - Application Programming Interface

AJAX – Asynchronous JavaScript and XML

DMZ – Demilitarized Zone

FK – Foreign Key

HTML – HyperText Markup Language

iOS – iphone Operating System

PK – Primary Key

pyOPENSSL- Python Open SSL Library Package

SSH – Secure Shell

SFTP – Secure File Transfer Protocol

SMS – Short Message Service

RDS - Radio Data System

VPC - Virtual Private Cloud

QR – Quick Response

WiFi -- Wireless Fidelity