

**Final Report: CHAI**

NET412

|  |  |
| --- | --- |
| Danny Yiu | 300675040 |
| Kenneth Sullivan | 300593020 |
| Marthe Nsaba | 300682252 |

Submitted to: Professor Hussein Fatmi

April 2, 2015

Table of Contents

Executive Summary 5

Mission Statement 5

Business Overview 5

Opportunities 5

Alternatives 5

Benefits 6

Financial Analysis 6

Product Summary 7

Costs 8

Development Roadmap 8

Risk Analysis 9

Patent Infringement Risks: 9

PCI Compliance: 9

Delays: 9

Vision Misunderstanding: 9

Frontend App Level: 9

Network Level: 9

Technical Documentation 10

Overview 10

Server First-Time Setup 11

Amazon EC2 Setup 11

Minimum Setup 11

Backend Installation 13

Transferring Files (SFTP) 13

SSH 13

Server File Structure 13

Server Configuration 14

Permissions 14

Apache Configuration 14

Automation 14

SSL 14

Websocket Initialization 15

Server Restart 15

Other Useful Notes 15

Importing Data (Optional) 16

Data Paths 16

Store Data 16

Menu Data 16

Images 16

Import Command 17

PayPal API Setup 17

API credentials 17

Merchant ID 17

Django Settings 18

Constants 18

Django URL patterns 18

URL Patterns 18

Django Model 19

Schema Command 19

Schema Diagram 19

Global Tables (Menu, Option, Category, Store, Customer) 20

Menu 20

Option 20

Category 20

Store 20

Customer 20

Store-specific Tables (Inventory, Order): 20

Inventory 20

Order 21

Order-specific Tables 21

OrderDetail 21

Web Service: Administration Page 21

Django Paths 21

Admin Account Creation (Commands) 21

Admin Login 22

Admin Function - Menu Management 22

Admin Function – Store Management 23

Admin Function – Store User Account 24

Web Service: Store API 25

Store Login Page (GET) 25

Store Landing Page (GET) 25

Inventory Item Active/Inactive Status Update (POST) 25

Order Complete Update (POST) 26

Store Employee Login (POST): 26

Store Employee Logout (POST): 27

Store QR Code Generation (GET): 27

Web Service: Customer API 28

New Order (POST): 28

New Account Registration (POST): 28

Menu Request (GET): 29

Product Options Request (GET): 29

PayPal Token Retrieval (GET): 30

Order Status Page (GET): 30

Front-end: Store Landing Page 31

Login 31

Store Landing Page 33

QR code page 35

Front-end: Customer App 36

Development Environment 36

Android Specific Notes 37

iOS Specific Notes 38

App Flow 39

Store Page: Before QR Code Scan 39

QR Scanner 40

Store Page: Menu 41

Store Page: Options 42

Store Page: Shopping Cart 43

Payment: PayPal 44

Order Status Page 45

Known Issues and Future Improvements 47

Admin Page 47

Store Landing Page 48

Customer App 48

Server 48

Glossary 49

References 49

# 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 code25 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.

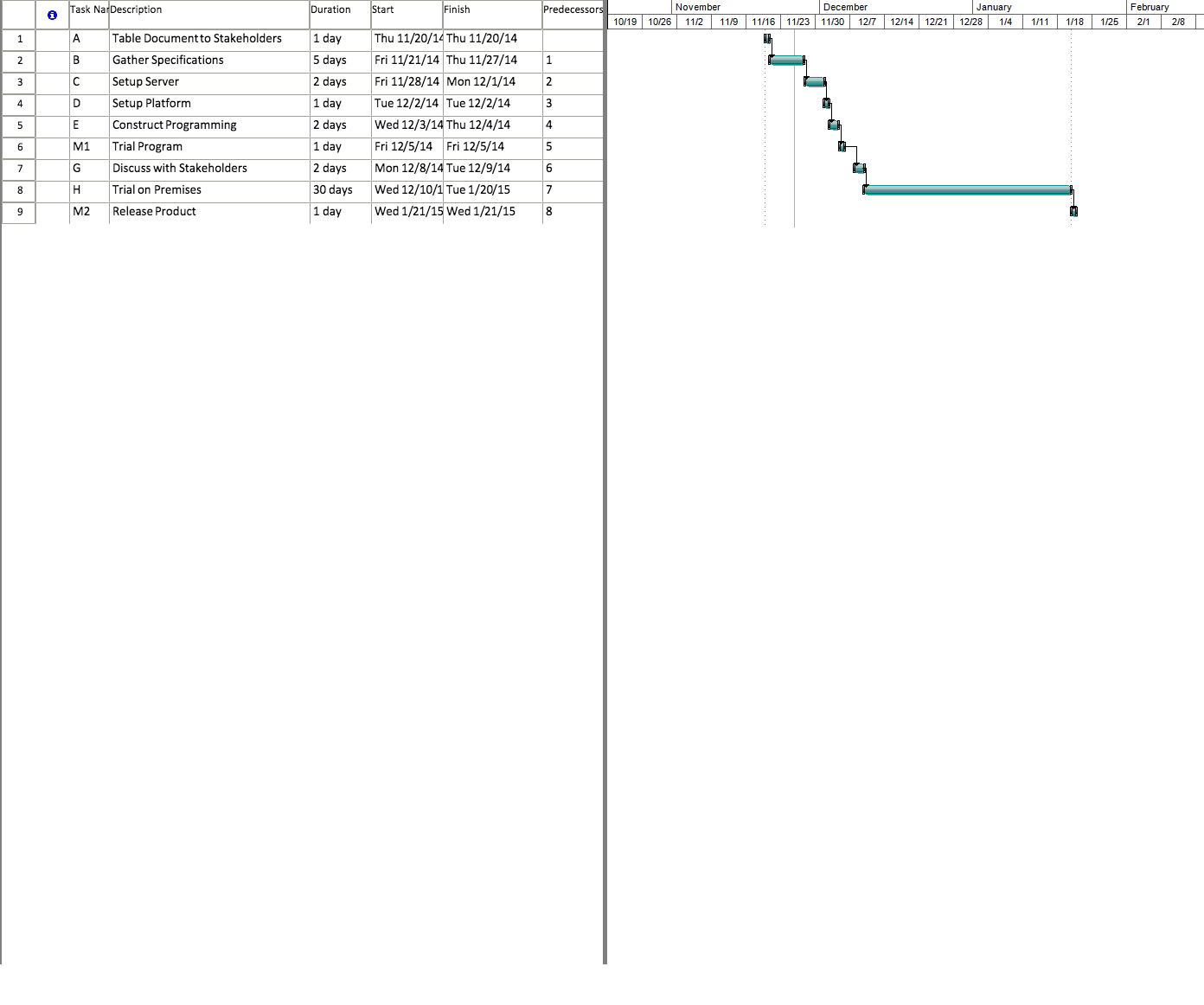
|  |  |
| --- | --- |
| **Component** | **Description** |
| User App | Function:  The front-end interface of the solution for customers at the coffee shop.  Capabilities:   * QR code scanning * PCI compliant transaction through PayPal * Secure connection with cloud service through SSL * Android and iOS 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 |

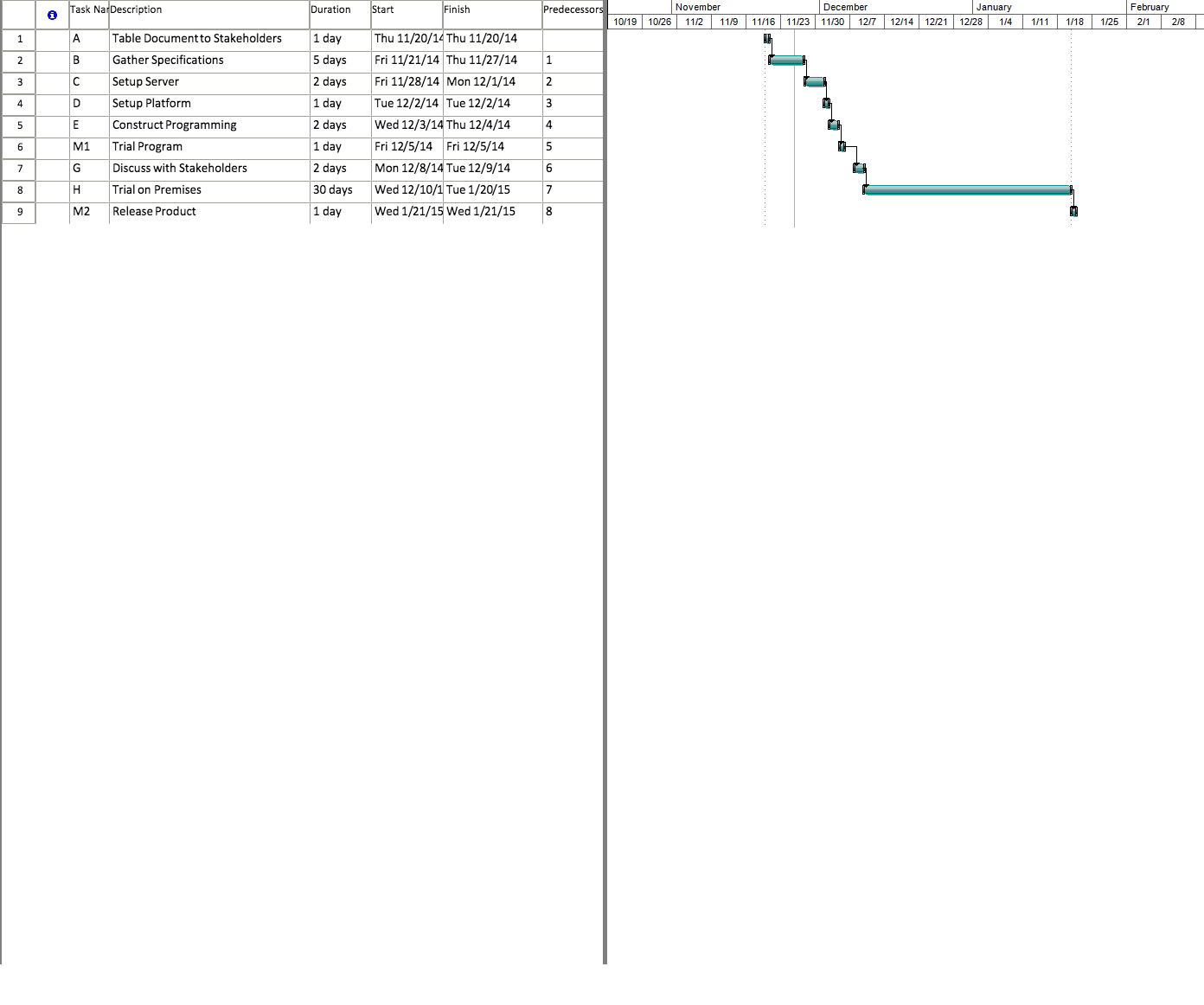
## Costs

|  |  |
| --- | --- |
| **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.*  PayPal credit card fees: *2.9% + $0.30 per transaction* |
| Support | Maintenance and upgrades of the system. Costs plans are flexible and scale with needs. |

## 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. All customer payment information is handed to PayPal’s PCI compliant system, requiring only a generated token on our side to handle all transactions.

### 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.

### 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. With SSL, CSRF tokens can also be used for session authentication to prevent hostile HTTP request building. 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.

# Technical Documentation

## Overview

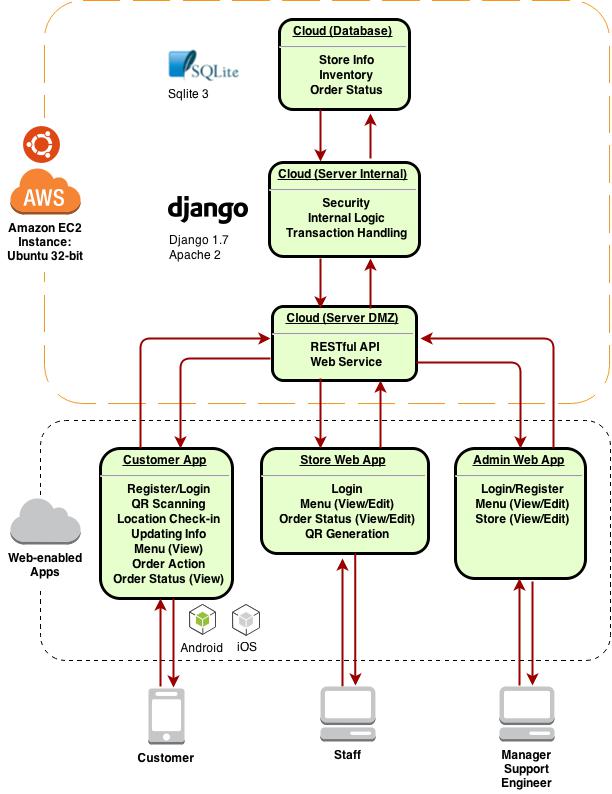


Figure : Logic flowchart

**Source code root:** <https://github.com/dannyyiu/410coffee/tree/master/src>

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 EC21 |
| Platform | 32bit Amazon Linux 2014.03 v1.0.7 running Python 2.72 |
| 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 SSH5 like any other hosting service.

**EC2 URL:** <http://aws.amazon.com/ec2/>

#### Minimum Setup

* Single t1.micro or t2.mirco instance6
* One Elastic IP7 attached
* Inbound rules allow ports: SSH (22), HTTP (80), Websocket8 (1025), HTTPS (443)
* One key pair9 attached for SSH/SFTP access

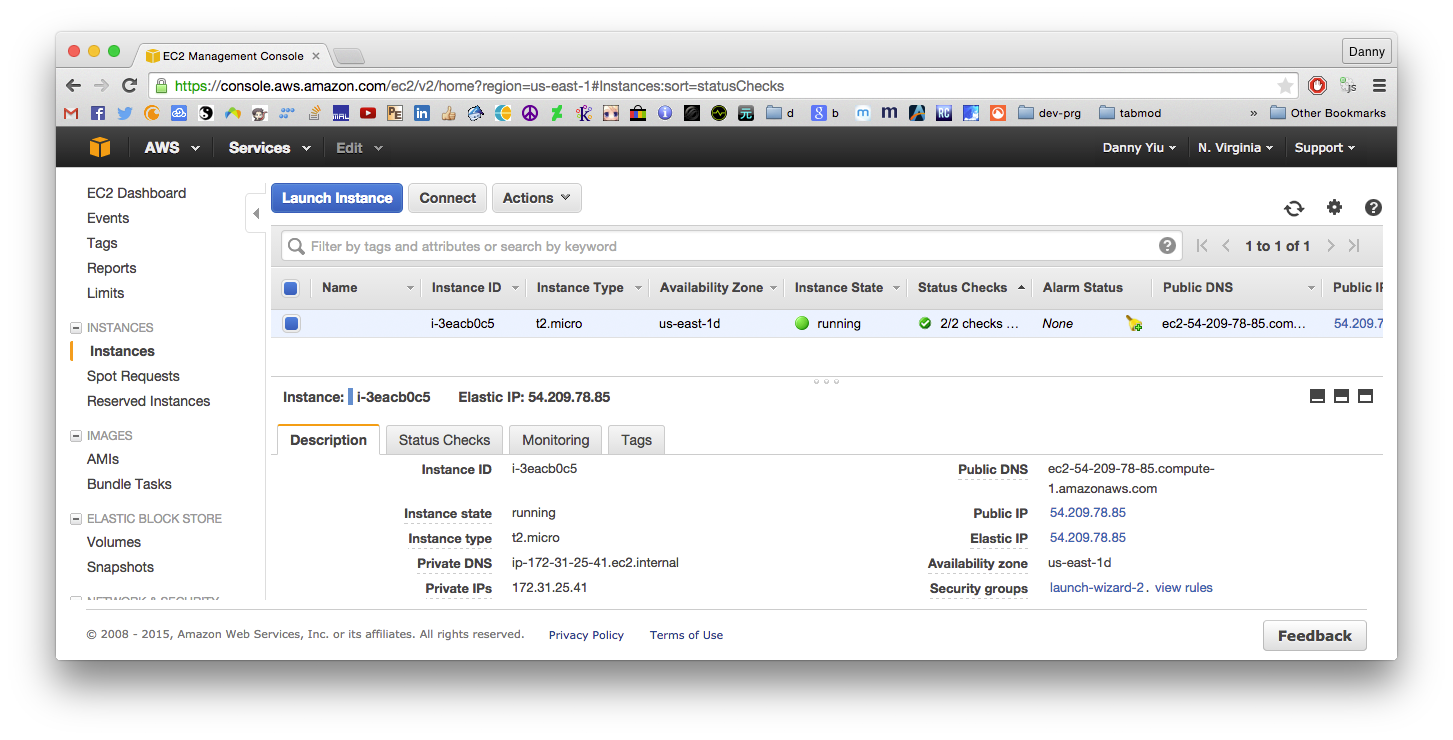


Figure : EC2 Instance

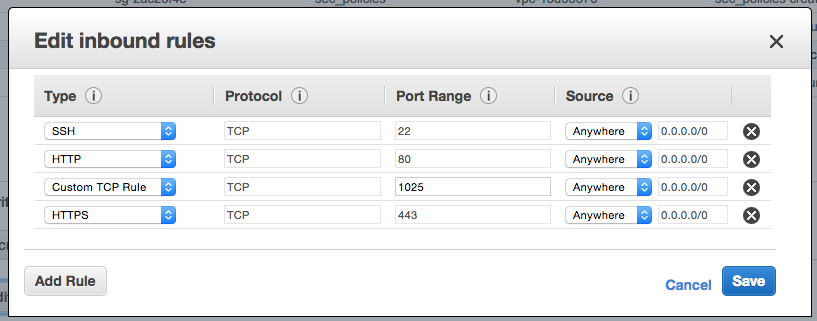


Figure : Inbound rules

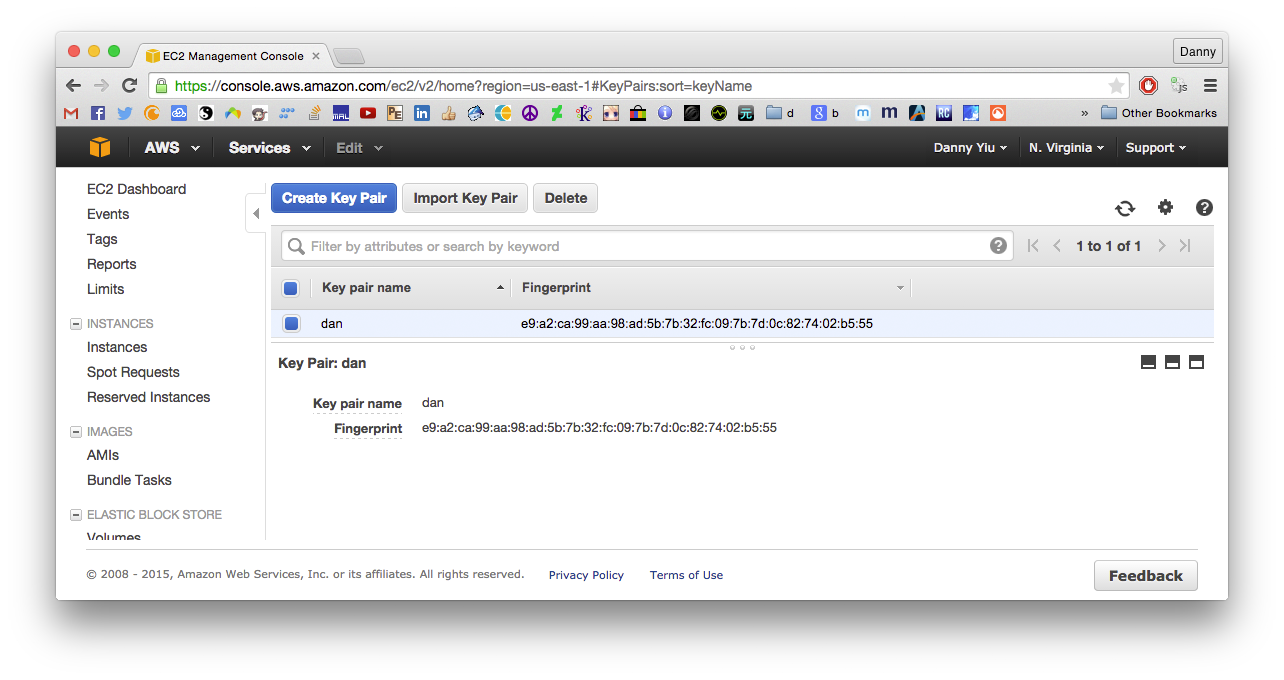


Figure : Keypair for SSH/SFTP. This is a .pem file used for authentication.

The Elastic IP is the public IP address of the instance, which can be used to update domain name DNS10 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”:

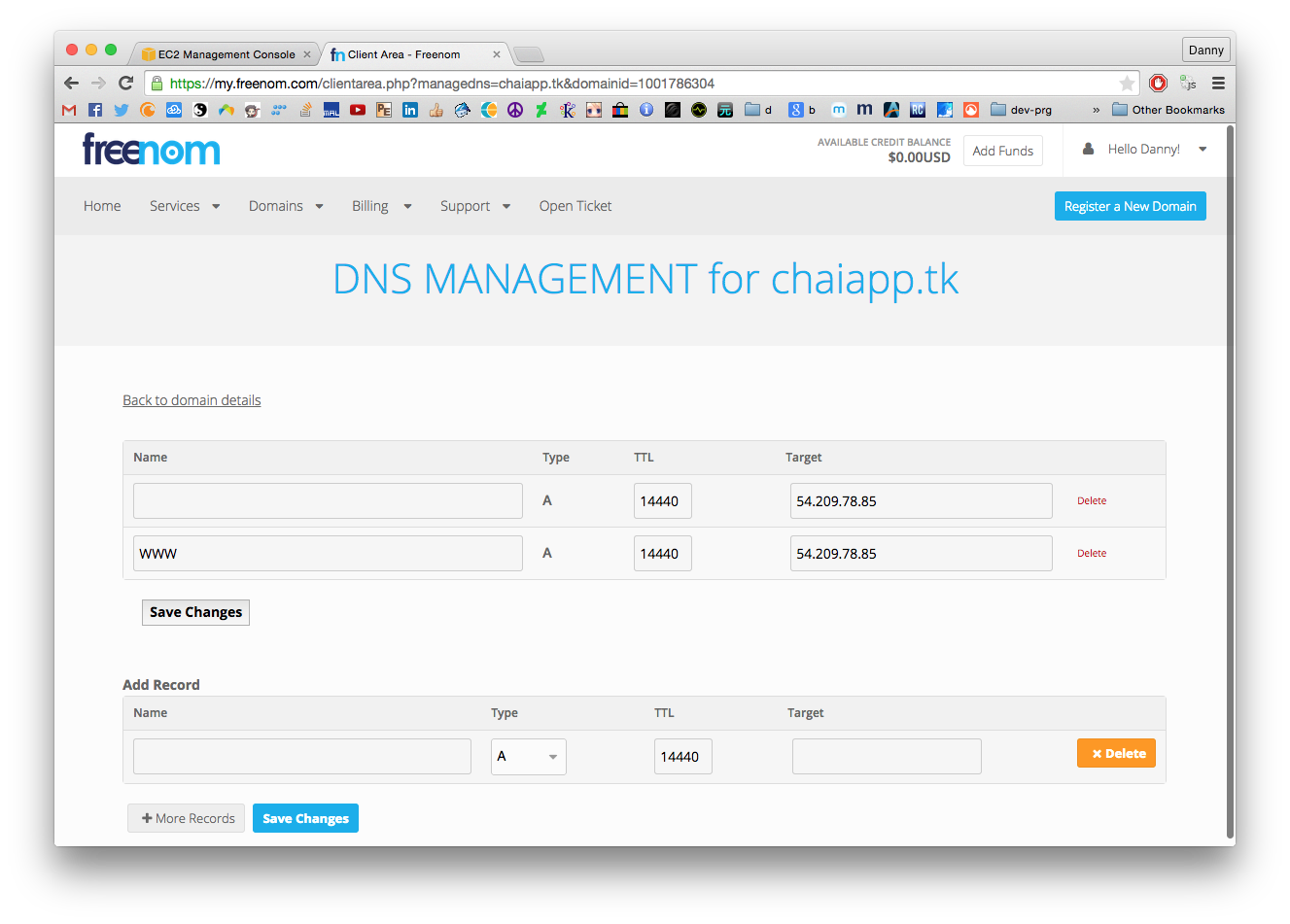


Figure : Updating DNS A records to add a domain name

### 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 private key 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.

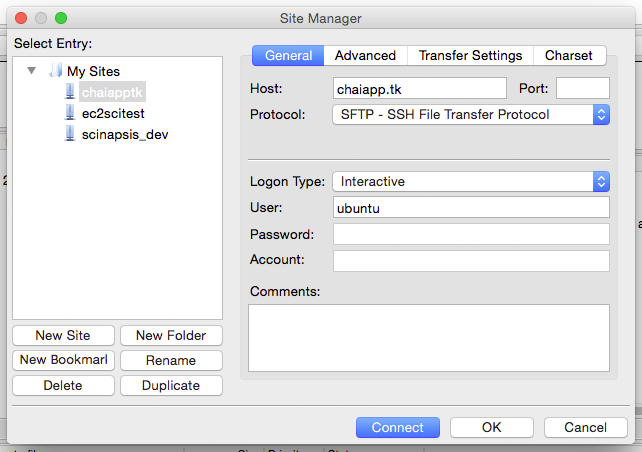


Figure : SFTP settings on Firezilla

#### SSH

Required:

* SSH client (SSH command in UNIX systems, or PuTTY11 for Windows)
* SSH private key 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](mailto: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

|  |  |
| --- | --- |
| ~/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\_cloud\_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\_cloud\_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 bundle12 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>)

Configuration Paths:

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

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/chai.config):

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

Enabling and disabling SSL can be done by commenting out the Listen commands in the ports.conf file (Lines 6-14):

Listen 80

<IfModule ssl\_module>

Listen 443

</IfModule>

<IfModule mod\_gnutls.c>

Listen 443

</IfModule>

#### Websocket Initialization

A websocket connection is required for all real-time updates, such as order status updates. This is a separate connection using port 1025 that is kept alive for every websocket request (ws:// or wss://) or listener. This is done through the Twisted library, hosting a websocket server separate of Apache. No additional Apache configurations are required as Twisted works in parallel.

File Paths:

|  |  |
| --- | --- |
| Websocket package | twisted-websocket-4173-4 |
| twistd path | ~/chai\_cloud\_deploy/\_scripts/twisted-websocket-4173-4/bin/twistd |
| twistd Python file | ~/chai\_cloud\_deploy/ws\_server.py |
| twistd store | ~/chai\_cloud\_deploy/twisted\_store/ |
| twistd PID file | ~/chai\_cloud\_deploy/twistd.pid |
| Websocket start script | ~/chai\_cloud\_deploy/\_ws-start |

For convenience, an automation script \_ws-start is included to start the websocket server:

cd ~/chai\_cloud\_deploy/

./\_ws-start

A PID of the websocket server process is generated at ~/chai\_cloud\_deploy/twistd.pid. This is used to keep track of the process in case it needs to be terminated. To stop the websocket server, run this command:

cd ~/chai\_cloud\_deploy/

sudo kill $(sudo cat twistd.pid)

#### Server Restart

Whenever there are new configurations, new static files, changes in Django Python files, or simply requiring a server refresh, the following commands are standard:

cd ~/chai\_cloud\_deploy/

./manage.py collectstatic

./manage.py syncdb

sudo service apache2 restart

#### Other Useful Notes

* To restart the server, run: sudo service apache restart
* Django debug messages and Apache error log: /var/log/apache2/error.log

### Importing Data (Optional)

We included a script to import your existing store and menu information programmatically into the database.

#### Data Paths

|  |  |
| --- | --- |
| Rebuild database script | ~/chai\_cloud\_deploy/\_scripts/\_rebuild-db |
| Store raw data | ~/chai\_cloud\_deploy/\_scripts/raw-data/stores.data |
| Menu raw data | ~/chai\_cloud\_deploy/\_scripts/raw-data/menu.data |

#### Store Data

Import your data with the following format and save it as stores.data in the path specified above.

<short name>|<full name>|<phone number>\n

...

*Note: The short name must be separated by “\_”.*

Example:

4040\_creditview\_rd|4040 Creditview Rd|(416) 218-9039

4228\_midland\_ave|4228 Midland Ave|(416) 218-9039

885\_britannia\_rd\_w|885 Britannia Rd W|(416) 218-9039

3650\_derry\_rd\_e|3650 Derry Rd E|(416) 218-9039

#### Menu Data

Import your data with the following format and save it as menu.data in the path specified above.

<item name>|<category>|<option>|<full price>|<image filename>|<description>\n

...

Example:

Decaf|coffee|small|1.60|coffee-decaf.png|All the great taste of your Tim Hortons Coffee, without the caffeine.

Decaf|coffee|reg|2.10|coffee-decaf.png|All the great taste of your Tim Hortons Coffee, without the caffeine.

Decaf|coffee|large|2.40|coffee-decaf.png|All the great taste of your Tim Hortons Coffee, without the caffeine.

#### Images

Using the image filenames in the previous section, you can import all your product images to be displayed on the customer’s app. All images must be size *160 x 160 pixels* for best results.

Menu images path:

~/chai\_cloud\_deploy/static/menu\_img/

To refresh the cloud service with the new images, run the following commands:

cd ~/chai\_cloud\_deploy/

manage.py collectstatic

sudo service apache2 restart

#### Import Command

Will all the data formatted as above, run the below commands to execute an automation script that migrates all the data into the database:

cd ~/chai\_cloud\_deploy/\_scripts/

./\_rebuild-db

sudo service apache2 restart

### PayPal API Setup

In order for the customer app to accept payment, a PayPal business account must be set up. The minimum requirements to use PayPal Classic API14 for our Express Checkout13 are:

* API username
* API password
* API signature
* merchant account ID

#### API credentials

URL: <https://developer.paypal.com/webapps/developer/applications/accounts>

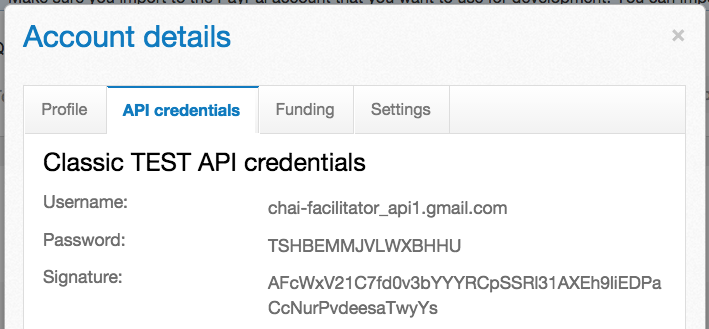


Figure : API username, password, signature

#### Merchant ID

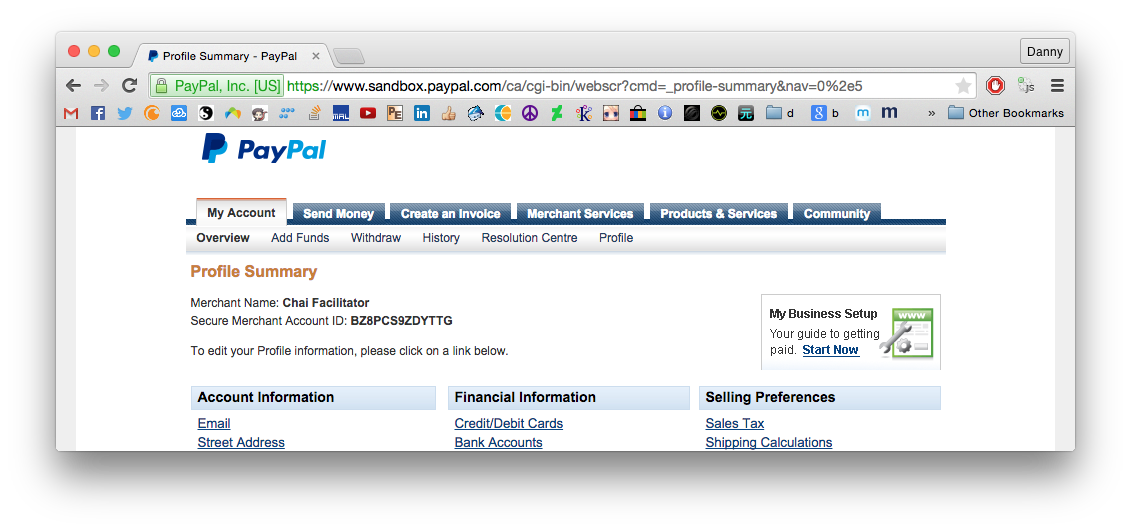
URL:<https://www.sandbox.paypal.com/ca/cgi-bin/webscr?cmd=_profile-summary&nav=0%2e5> 

Figure : Merchant account ID

## Django Settings

Most settings are ready to work out of the box (the ones requiring change are addressed above in the “Server First-time Setup” section). Below are some specifications describing this file.

**Path:** ~/chai\_cloud\_deploy/chai\_cloud/settings.py

### Constants

|  |  |
| --- | --- |
| HTTP\_URL | Address when calling self-origin URLs. In most cases, set as localhost. |
| WS\_URL | Websocket server public address |
| PP\_CURRENCY | PayPal API currency code. See: <https://developer.paypal.com/docs/classic/api/currency_codes/> |
| PP\_TOKEN\_URL | PayPal API endpoint URL. See: <https://developer.paypal.com/docs/classic/api/endpoints/> |
| PP\_EMAIL | PayPal API merchant API email (not login email, see PayPal API section) |
| PP\_MERC\_ID | PayPal API merchant ID |
| PP\_PSS | PayPal API merchant password (not login password, see PayPal API section) |
| PP\_SIGNATURE | PayPal API merchant signature |
| PP\_STORE\_NAME | PayPal API store name displayed during checkout (NVP format, see PayPal API section) |
| ROOT\_URLCONF | URL definitions file (urls.py) |
| TIME\_ZONE | Django Time zone code. |
| STATIC\_ROOT | Static files root directory to be hosted by Apache. All static files will be copied to this destination upon command manage.py collectstatic |
| STATIC\_URL | URL path for public access to static files |
| STATIC\_ROOT | Static files root directory to be copied to and hosted by Apache |
| STATICFILES\_DIRS | Static files directories to be included during collectstatic. All contents in these folders will be copied to STATIC\_ROOT during manage.py collectstatic |
| SECURE\_PROXY\_SSL\_HEADER | SSL required. Sets header for HTTPS connections. |
| SESSION\_COOKIE\_SECURE | SSL required. Marks cookies as secure, for browsers to send only under HTTPS conditions. |
| CSRF\_COOKIE\_SECURE | SSL required. Marks the CSRF cookie as secure, for browsers to send only under HTTPS conditions. |

## Django URL patterns

URL patterns are set in Django to map URL Regular Expressions15 to functions in views.py, which returns some form of HTTP Response, be it a JSON string or a fully rendered webpage. Below are some specifications describing the URL patterns.

**Path:** ~/chai\_cloud\_deploy/chai\_cloud/urls.py

### URL Patterns

|  |  |
| --- | --- |
| admin/ | Renders an admin landing page, and all other admin sub-URLs. Handled by Django default admin view. (HTML) |
| store-<store-name>/ | Store landing page, or store API; Depends on request type. (HTML for GET, JSON for POST) |
| s-qr/ | QR code generation API (HTML with QR image) |
| c-order/ | Customer API: New orders (JSON) |
| c-menu/ | Customer API: Get store menu (JSON) |
| c-prod/ | Customer API: Get product options (JSON) |
| c-token/ | Customer API: PayPal SetExpressCheckout (JSON) |
| c-status-<store-name>/ | Customer order status page (HTML) |
| c-register | Customer API: Register new account |
| e9b14fld.htm | RapidSSL verification |
| static/ | Static files (CSS, images...)\_ |

## Django Model

The model for Django is the database structure, representing the Model part of the Django MVC16. We chose to use sqlite3 for the implementation to simplify deployment. Sqlite3 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.

**Database path:** ~/chai\_cloud\_deploy/db.sqlite3

### Schema Command

sqlite3 ~/chai\_cloud\_deploy/db.sqlite3

.schema

### Schema Diagram

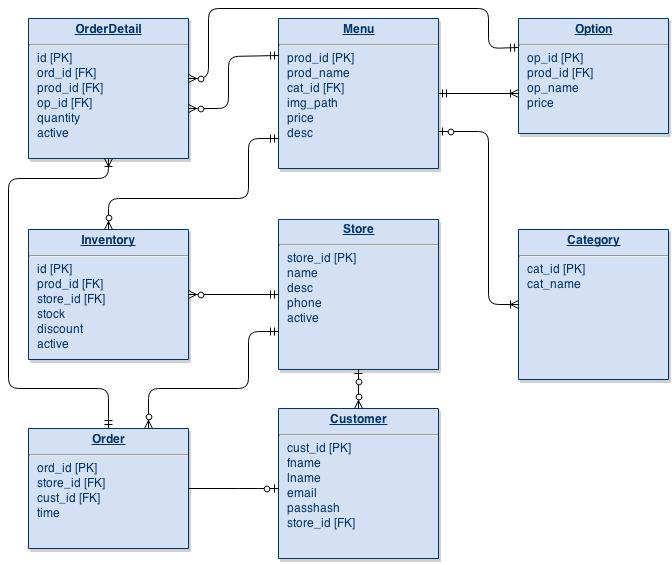


Figure : Database schema

Our model is heavily optimized for our API17 calls though use of foreign key references. The tables are linked together in a way that most API calls require at most two queries. The organization of the tables can be split into three categories:

### Global Tables (Menu, Option, Category, Store, Customer)

Global tables are tables with static information not affected by the stores or new orders. They contain information globally relevant to all API calls. For example, a customer’s name stays constant everywhere.

#### Menu

|  |  |
| --- | --- |
| prod\_id [PK] | Product ID |
| prod\_name | Product name |
| cat\_id [FK] | Category ID, references Category table |
| img\_path | Name of product image file |
| price | Base price of item |
| desc | Product description |

#### Option

|  |  |
| --- | --- |
| op\_id [PK] | Option ID |
| prod\_id [FK] | Product ID, references Menu table |
| op\_name | Category ID, references Category table |
| price | Price difference of option to the base price |

#### Category

|  |  |
| --- | --- |
| cat\_id [PK] | Category ID |
| cat\_name | Category name |

#### Store

|  |  |
| --- | --- |
| store\_id [PK] | Store ID |
| name | Store short name |
| desc | Store description (Long name) |
| phone | Phone number |
| active | Active status (1 or 0) |

#### Customer

|  |  |
| --- | --- |
| cust\_id [PK] | Customer ID |
| fname | First name |
| lname | Last name |
| email | Email address |
| passhash | Bcrypt hash value of password |
| store\_id[FK] | Store ID, references Store table |

### Store-specific Tables (Inventory, Order):

These tables contain information unique to each store. For example, two stores cannot share the same inventory, one store cannot have an order from another store.

#### Inventory

|  |  |
| --- | --- |
| id [PK] | Inventory ID |
| prod\_id [FK] | Product ID, references Menu table |
| store\_id [FK] | Store ID, references Store table |
| stock | Stock amount per item |
| discount | Discount price modifier |
| active | Inventory item active status |

#### Order

|  |  |
| --- | --- |
| ord\_id [PK] | Order ID |
| store\_id [FK] | Store ID, references Store table |
| cust\_id [PK] | Customer ID, references Customer table |
| time | Date and time of order |

### Order-specific Tables

These tables contain information unique to each order. For example, an order from one store cannot have quantities from another store’s order.

#### OrderDetail

|  |  |
| --- | --- |
| id [PK] | Inventory ID |
| ord\_id [FK] | Order ID, references Order table |
| prod\_id [FK] | Product ID, references Menu table |
| op\_id [FK] | Option ID, references Option table |
| quantity | Quantity of item ordered |
| active | Active orders are not completed |

## Web Service: Administration Page

The administration page is for top-level managers and engineers to add, delete and change store and menu information, as well as managing staff accounts.

### Django Paths

|  |  |
| --- | --- |
| Admin URL declarations | ~/chai\_cloud\_depoly/chai\_cloud/urls.py |
| Admin template files | ~/chai\_cloud\_deploy/templates/admin/ |

*Note: The current version uses the default Django admin template with minor adjustments. For more template customization, edit the template files in the template files path indicated in the table above.*

### Admin Account Creation (Commands)

For managers to manage stores and menu items, an admin account is required. This account allows a manager to login to a management page to edit top-level franchise information. Admin accounts are created directly with Django’s manage.py createsuperuser command. User email and password will be prompted after this command. The below commands will set up an admin account and restart the server:

cd ~/chai\_cloud\_deploy/

./manage.py createsuperuser <username>

./manage.py syncdb

sudo service apache2 restart

### Admin Login

The default landing page for admin accounts is the login page if not logged in.

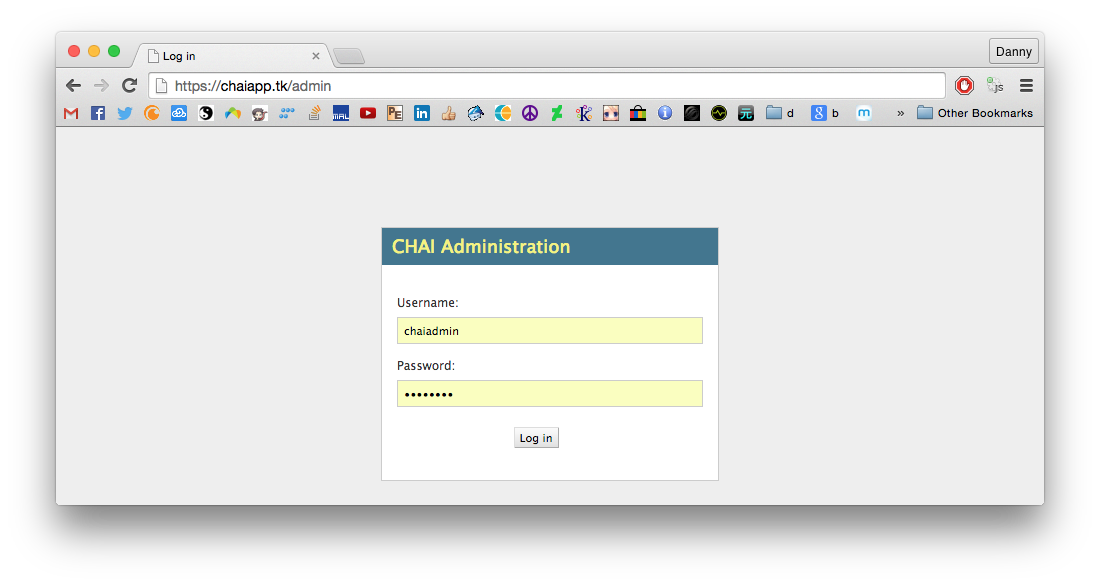
**Login URL:** https://<domain-name>/admin  


Figure : Admin login page

### Admin Function - Menu Management

Admin users can add, delete, and edit menu items through the following pages.

**Database table involved:** Menu

**Menu URL:** https://<domain-name>/admin/api/menu/

****

Figure : Menu management – add/detele menu items

**Options management URL:** https://<domain-name>/admin/api/menu/<prod-id>/

**Database tables involved:** Menu, Option

**Django definition:** ~/chai\_cloud\_deploy/api/admin.py (Lines 17-24)

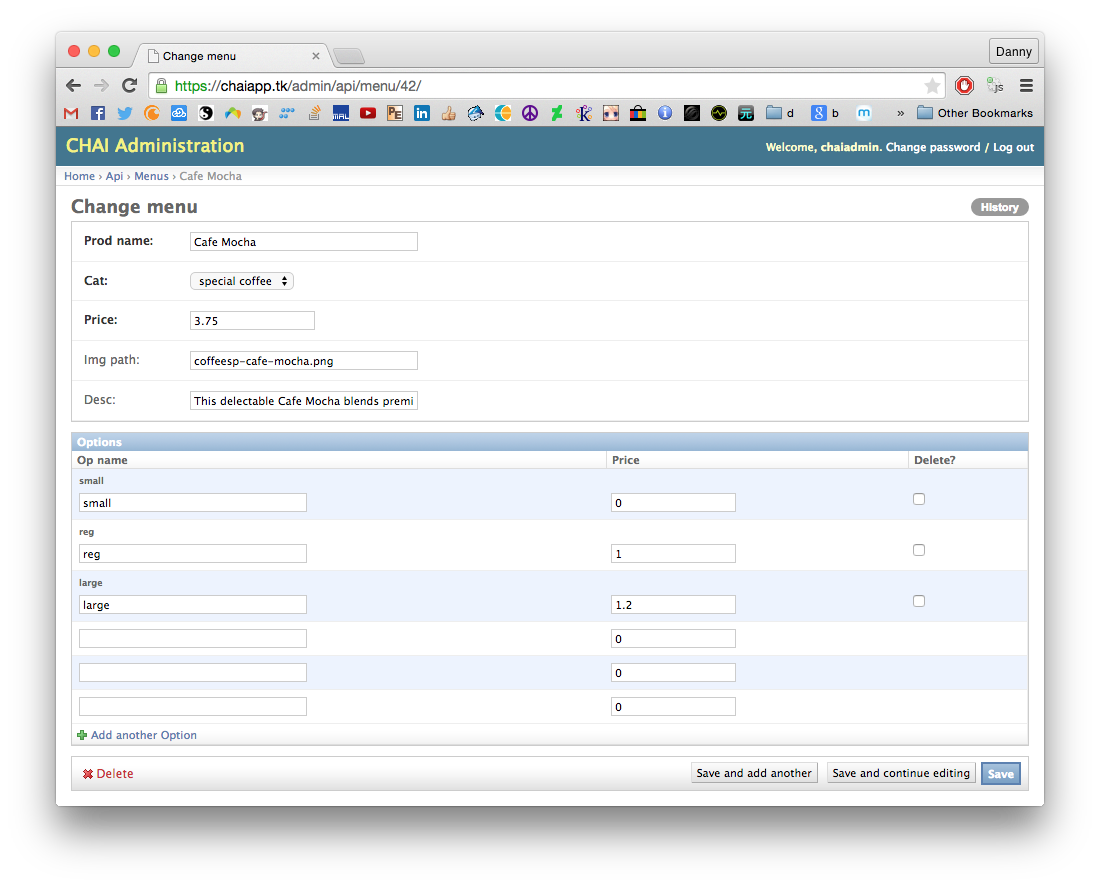


Figure : Options management – add/delete/change options for each item

### Admin Function – Store Management

Admin users can add, delete, and edit stores through the following pages.

**Store management URL:** https://<domain-name>/admin/api/store/

**Database tables involved:** Store

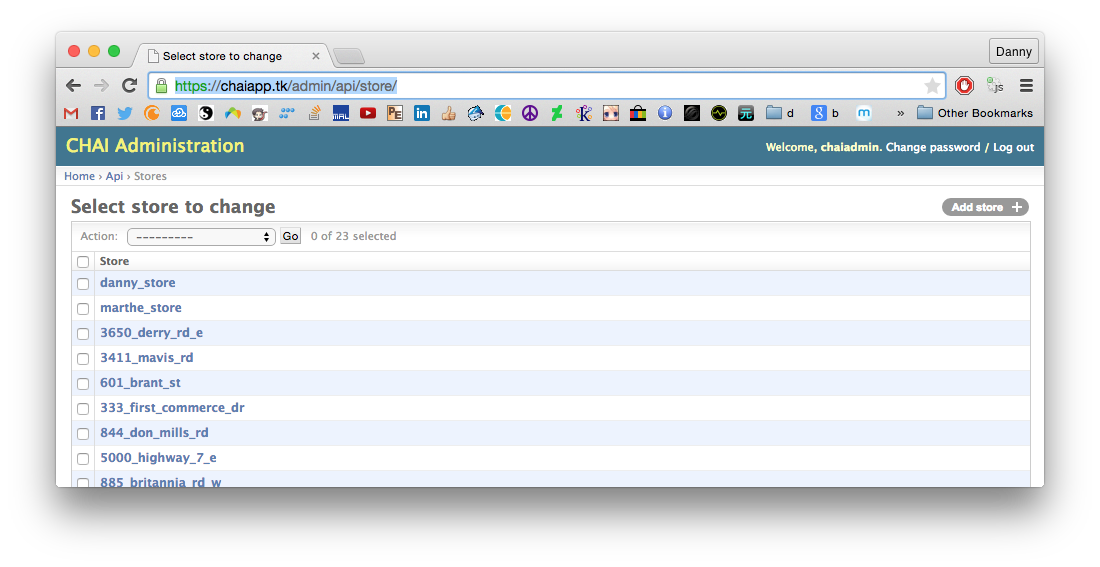


Figure : Stores management: - add/delete stores

**Store details management URL:** https://<domain-name>/admin/api/store/<store-id>/

**Database tables involved:** Store, Inventory

**Django definition:** ~/chai\_cloud\_deploy/api/admin.py (Lines 5-24)

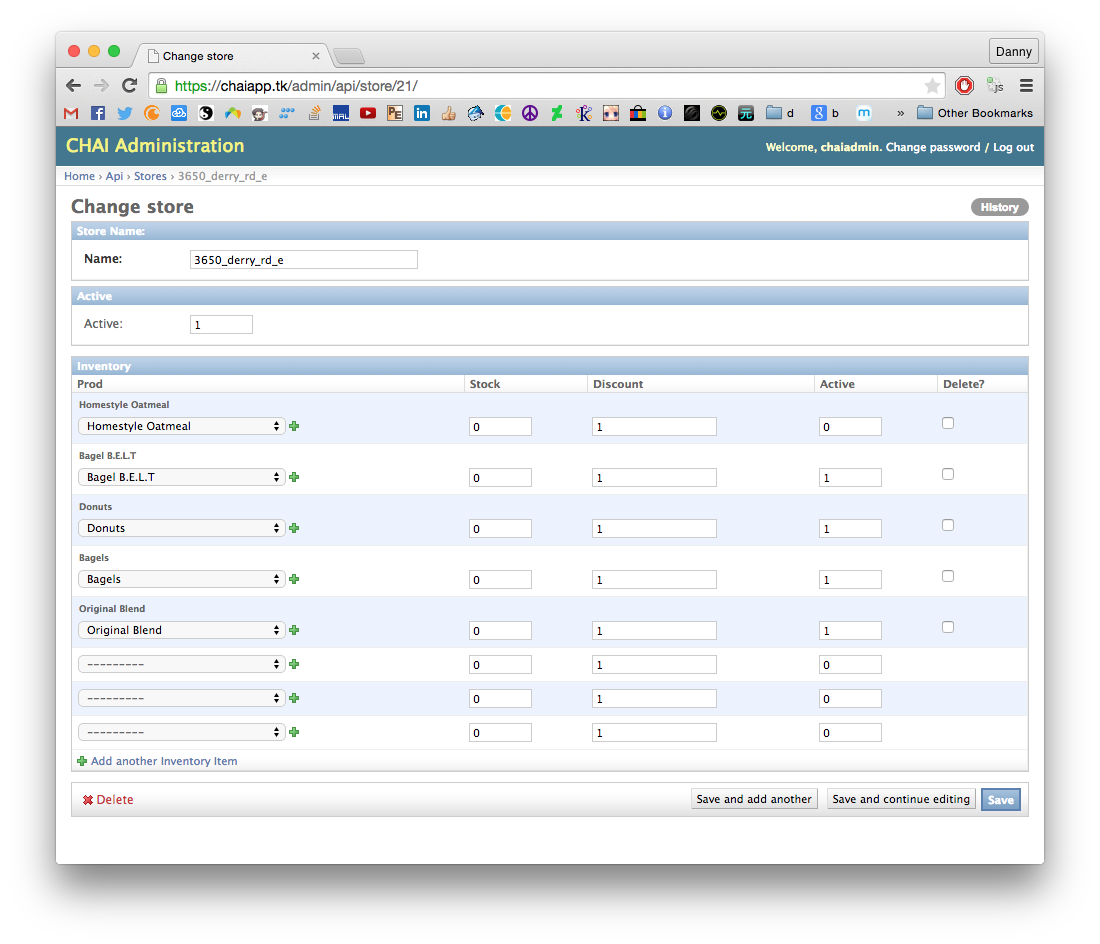


Figure : Individual store management – add/delete inventory items, change active status, change store name for each individual store

### Admin Function – Store User Account

To ensure staff-only login to store pages, user authentication is required for all store interfaces. However, store staff should not have admin privileges, which is automatically granted through createsuperuser command. Therefore, these users must be created by an admin account.

**Users URL:** https://<domain-name>/admin/auth/user/

**Database tables involved:** auth\_user (Django default)

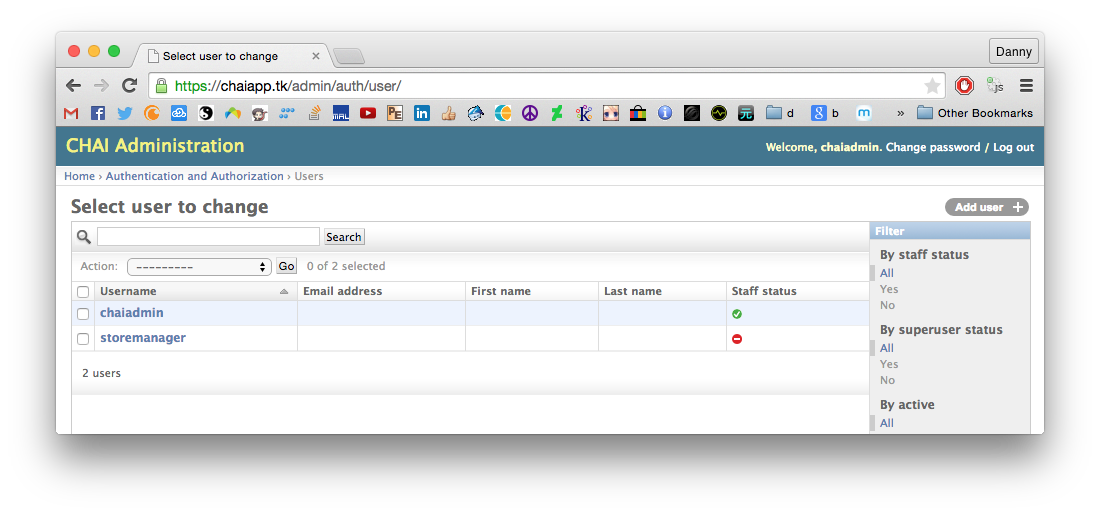


Figure : Manage staff users

## Web Service: Store API

The Web Service API is the outer-most layer, serving as the Demilitarized Zone (DMZ) of our cloud service. For users to access our cloud, all requests by apps must be done through this layer. All Web Services are RESTful18: HTTP GET/POST requests sent to the webserver will respond with data. The data can either be JSON20 strings or rendered webpages. Below are the API methods intended for store staff.

### Store Login Page (GET)

**Overview:**

This page is intended for store employees. Initially when user is not logged in, a login template will be rendered and returned as HTML prompting login. Full interface description in the “Frontend: Store” section below.

**URL:**store-<store\_name>/

**Request type:** GET

**Response type:**HTML

**Parameters:**N/A

**Logic pseudocode** (views.py Lines 50-68)*:*

# Check user login status

if request.method == ‘GET’ and not request.user.isauthenticated():

# if not logged in, render login page

return render(store\_login\_page)

### Store Landing Page (GET)

**Overview:**

This page is intended for store employees. If user is detected as logged in, the store landing page is rendered and returned as HTML. Full interface description in the “Frontend: Store” section below.

**URL:**store-<store\_name>/

**Request type:** GET

**Response type:**HTML

**Parameters:**N/A

**Logic pseudocode**(views.py Lines 70-73)

# Check user login status

if request.method == ‘GET’ and request.user.isauthenticated():

# Get some information from database...

database\_json = get\_some\_json\_data()

# Render into store page

return render(store\_landing\_page, database\_json)

### 

### Inventory Item Active/Inactive Status Update (POST)

**Overview:**

Inventory items be marked as active or inactive by employees on the store landing page. This allows the store to disable items out of stock, or enable items with new stock arrival. The active/inactive status determines which inventory items are displayed on the customer’s app. This API is intended for internal AJAX19 calls from the “Active”/“Inactive” button events triggered on the store landing page.

**URL:**store-<store\_name>/

**Request type:** POST

**Response type:**JSON (content insignificant)

**Parameters:**

‘active’ : String, ‘1’ or ‘0’

‘prod\_id’ : String, product ID

‘store\_id’ : String, store ID

**Logic pseudocode**(views.py Lines 80-91)*:*

# Confirm button event ‘active’

if request.POST.get(‘active’):

# Get database selector for selected item in the HTML form

selected = database\_get(html\_form\_selected)

# Update selector with new active status

selected.update(active=request.POST[‘active’])

# Return JSON string HTTP response

return HttpResponse(json\_string(selected))

### Order Complete Update (POST)

**Overview:**

Customer orders can be marked as complete by employees on the store landing page. This allows the customer to view their order status as complete in real-time. This API is intended for internal AJAX calls from the “Complete” button events triggered on the store landing page.

**URL:**store-<store\_name>/

**Request type:** POST

**Response type:**JSON (content insignificant)

**Parameters:**Form data

‘complete\_order’ : String, ‘Complete’

‘details\_id’ : String, order details ID

**Logic pseudocode**(views.py Lines 95-102)*:*

# Confirm button event ‘complete\_order’

if request.POST.get(‘complete\_order’):

# Get database selector for selected order in the HTML form

selected = database\_get(html\_form\_selected)

# Mark selected order as inactive (completed = inactive)

selected.update(active=0)

update\_websocket() # Update websocket for listening customers

# Return JSON string HTTP response

return HttpResponse(json\_string(selected))

### 

### Store Employee Login (POST):

**Overview:**

Before user in detected as logged in, a login page is rendered. This login page sends username and password as a login request to this API method. Once the username is checked as valid and active, and password is verified as correct, user will be logged in and redirected to the store landing page. If user is account is disabled or password is incorrect, the login page will be rendered again with the appropriate warning message. This API is intended for internal AJAX calls from the “Login” button event triggered on the store login page.

**URL:**store-<store\_name>/

**Request type:** POST

**Response type:**HTML

**Parameters:**Form data

‘username’ : String, store employee username

‘passw’ : String, store employee password

**Logic pseudocode** (views.py Lines 105-127)*:*

# Confirm POST type and button event ‘active’

if request.POST.get(‘username’) and request.POST.get(‘passw’):

username = request.POST[‘username’]

passw = request.POST[‘passw’]

# Check authentication

user = authenticate(username, passw)

if user:

# Password correct

if user.is\_active:

# User active

return redirect(store\_landing\_page)

else:

# User inactive

return render(login\_page, “Inactive account”)

else:

# Password incorrect

return render(login\_page, “Incorrect password”)

### Store Employee Logout (POST):

**Overview:**

Logged in user will be logged out. All other users will have no effect. User information is gathered from session cookies. This API is intended for internal AJAX calls from the “Logout” button event triggered on the store landing page. As the store landing page with no logged in user automatically renders the login page again, there is no need to render any page.

**URL:**store-<store\_name>/

**Request type:** POST

**Response type:**JSON (content insignificant)

**Parameters:**URL parameters

‘logout’ : String, “1”

**Logic pseudocode** (views.py Lines 130-132)*:*

# Confirm POST type and button event ‘active’

if request.POST.get(‘logout’):

logout(request)

return HttpResponse(json\_string(logout\_confirmation))

### Store QR Code Generation (GET):

**Overview:**

A unique QR code can be generated for each store based on the store ID. This API method takes generates a QR code and renders a HTML page containing the QR code image.

**URL:**s-qr/

**Request type:** GET

**Response type:**HTML

**Parameters:**URL parameters

‘store\_id’ : String, store ID

**Logic pseudocode** (views.py Lines 137-154)*:*

# Confirm GET type and button event ‘active’

if request.GET.get(‘store\_id’):

return render(qr\_page, request.GET[‘store\_id’])

## Web Service: Customer API

Customer app interacts with the cloud service through a set of APIs designed for customers. Below are the API methods intended for the customer app.

### New Order (POST):

**Overview:**

Adds a new order to the system based on the customer’s successful transaction. Although this is a customer API, it is not called directly by the customer app. Instead, it is called by the cloud after transaction API has confirmed payment. The order is added to the cloud’s database, showing up in real time to both the store landing page and the customer’s order status page through a websocket update.

**URL:**c-order/

**Request type:** POST

**Response type:**HTML

**Parameters:**URL parameters

‘order\_list’ : String, format: "[{'prod\_id':2, 'op\_id':1},...]"

‘store\_name’ : String, store name

‘email’ : String, customer email used for payment and account

**Logic pseudocode** (views.py Lines 171-246)*:*

# Check parameters

if request.POST.get(‘order\_list’) and request.POST.get(‘store\_name’):

# Add order to database

database.udpate(

request.POST['order\_list'],

request.POST['store\_name'])

# Get last update information

details = json(get\_last\_db\_update)

# Websocket update

update\_websocket(details)

# Return JSON of details

return HttpResponse(json\_str(details))

### 

### New Account Registration (POST):

**Overview:**

Register new customer account to the cloud system. This is called by voluntary registration of customers, an optional API function only required when customer registration is enabled on the customer app. Passwords are saved using Python’s bcrypt encryption library.

**URL:**c-register/

**Request type:** POST

**Response type:**JSON (content insignificant)

**Parameters:**URL parameters

‘email’ : String, email address

‘pass’ : String, password

‘fname’ : String, first name

‘lname’ : String, last name

**Logic pseudocode** (views.py Lines 261-281)*:*

# Check parameters

if request.GET.get(‘order\_list’) and request.GET.get(...:

# Hash password

hashed = bcrypt.hash(request.GET['pass'])

# Update in database

database.customer.update(

request['fname'],

...,

hashed)

return HttpResponse(json\_string({'status': 'registered'}))

### Menu Request (GET):

**Overview:**

The customer app can retrieve store inventory information of any store with their store ID.

**URL:**c-menu/

**Request type:** GET

**Response type:**JSON

{ category\_name :

{ prod\_name :

{ 'img\_url' : img\_url,

'prod\_id' : prod\_id,

'prod\_name' : prod\_name,

}, ...

}, ...

}

**Parameters:**URL parameters

‘store\_id’ : String, email address

**Logic pseudocode** (views.py Lines 261-281)*:*

# Check parameters

if request.GET.get('store\_id'):

# Get inventory

inventory\_selected = database.inventory.get(

store\_id = request.GET['store\_id'])

# Format json

output = json\_format\_menu(inventory\_selected)

return HttpResponse(json\_string(output))

### Product Options Request (GET):

**Overview:**

The customer app can retrieve product options information of any product with their product ID.

**URL:**c-prod/

**Request type:** GET

**Response type:**JSON

{ op\_id :

{ op\_name : price }, ...

}

**Parameters:**URL parameters

‘prod\_id’ : String, email address

**Logic pseudocode** (views.py Lines 261-281)*:*

# Check parameters

if request.GET.get('prod\_id'):

# Get product options

options\_selected = database.options.get(

prod\_id = request.GET['prod\_id'])

# Format json

output = json\_format\_menu(options\_selected)

return HttpResponse(json\_string(output))

### PayPal Token Retrieval (GET):

**Overview:**

The customer app enters the PayPal Express Checkout page during their checkout process. To enter this page, an call to PayPal’s SetExpressCheckout21 API method must be called to retrieve a token. Since this call requires merchant credential information, it is safer to do the call on the cloud instead of the customer app. The customer app will call this API which in turns calls the PayPal SetExpressCheckout function and returns the token response back to the customer app. Return URL for the token is set to the order status page with the order details parameter.

**URL:**c-token/

**Request type:** GET

**Response type:**JSON

{ ‘PP\_SETCHECKOUT\_RESPONSE’: token }

**Parameters:**URL parameters

‘amt’ : String, total payment amount

‘orderdetails’ : String, json string for order details.

‘store\_id’ : String, first name

**Logic pseudocode** (views.py Lines 354-389)*:*

# Check parameters

if request.GET.get('amt') and ...:

# Build URL parameters with GET and PayPal parameters

params = build\_url\_params(request.GET)

# Call SetExpressCheckout API

token\_request = urlopen(setexpresscheckout\_url + params)

return HttpResponse(

{'PP\_SETCHECKOUT\_RESPONSE': token\_request.read()})

### 

### Order Status Page (GET):

**Overview:**

After payment is complete, the customer is redirected to the return URL to display the order status. This redirect includes the token and the order information, and calls this API function. First, a GetExpressCheckoutDetails22 API call is made to PayPal, checking the payment status. If “PaymentActoinNotInitiated” is returned, it means the payment button is pressed but is awaiting transaction completion by the merchant. If this is the case, a DoExpressCheckoutPayment23 API call is made to PayPal to finish the transaction. If “PaymentActionCompleted” (or NVP24 variable “ACK” = “Success”) is found in the return JSON is found, then the payment is verified. Only verified payments will be render the order status page.

**URL:**c-status-<store\_name>

**Request type:** GET

**Response type:**HTML

**Parameters:**URL parameters

‘token’ : String, PayPal token generated from SetExpressCheckout call

‘PayerID’ : String, PayPal payer ID.

**Logic pseudocode** (views.py Lines 354-389)*:*

# Check parameters

if request.GET.get('token') and request.GET.get('PayerID'):

# Call GetExpressCheckoutAPI

details\_request = pp\_get\_checkout\_details(token)

if "PaymentActionNotInitiated" in details\_request:

amt = details\_request['AMT']

# Call DoExpressCheckout

do\_checkout = pp\_do\_checkout(payer\_id, amt, token)

# Check sucesss

if do\_checkout['ACK'] == "Success":

# Auto register customer if not exist, for order history

customer\_sync(fname, lname, email)

# Call new order

urlopen('localhost/c-order' + params)

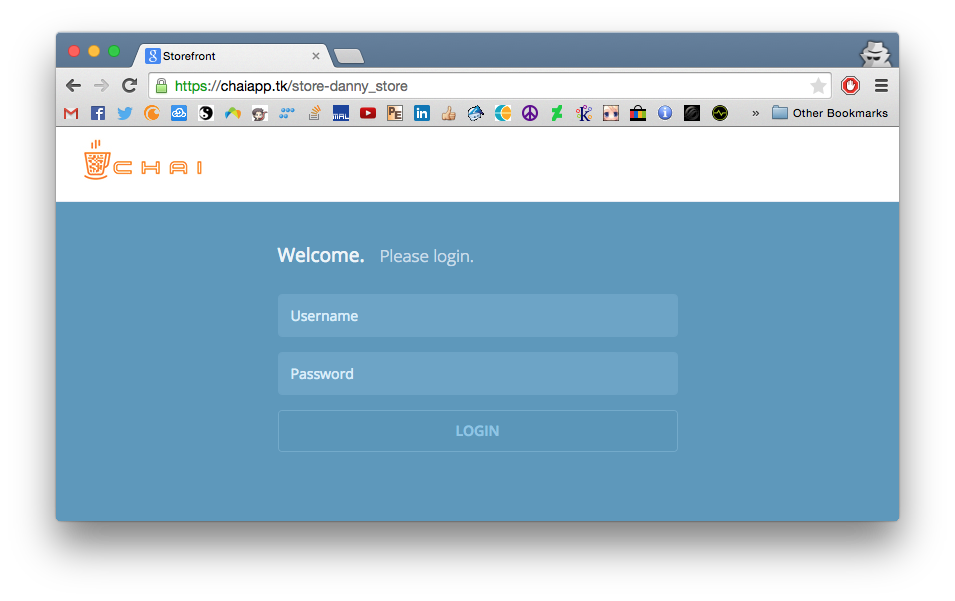
return render(order\_status\_page)

## 

## Front-end: Store Landing Page

Store employees interact with the cloud using the store landing page. This is the front-end interface designed to show order status and inventory with some limited updating functionality. It also interacts in real-time with the customer’s order status page, and determines what is displayed in the customer app’s menu.

### Login



**CSS theme used:** Metronic v3.6.1 (<http://keenthemes.com/>)

**Django variables:** alert: (String) alert box text

**Key Elements:**

|  |  |
| --- | --- |
| Username | Form input, id=“username” |
| Password | Form input, id=“passw” |
| Alert box | Button in div, class=“alert display-hide” (stays hidden by default) |
| Login Button | Form submit button, parent form class=“login-form” |

**Events pseudocode** (~/chai\_cloud\_deploy/templates/api/store-login.html, Lines 97-109)**:**

// Check of alerts in request parameters to show on default

if ("{{ alert }}" != "") {

// Remove the hidden class attribute to show alert

$(".alert").removeClass("display-hide")

// Replace text with alert text

$(".alert").text("{{ alert }}")

};

// Submit button event

$('.login-form').submit(function(event) {

// Get username and password

var u\_val = $(this).find("#username").val()

var p\_val = $(this).find("#passw").val()

// Field validation

if (u\_val && p\_val) {

// Password and username are both present

return; // Continue to submit

} else {

// Username and/or password missing

// Remove class with hide attribute to show alert box

$(".alert").removeClass("display-hide")

// Replace alertbox text

$(".alert").text("Please enter both username and password");

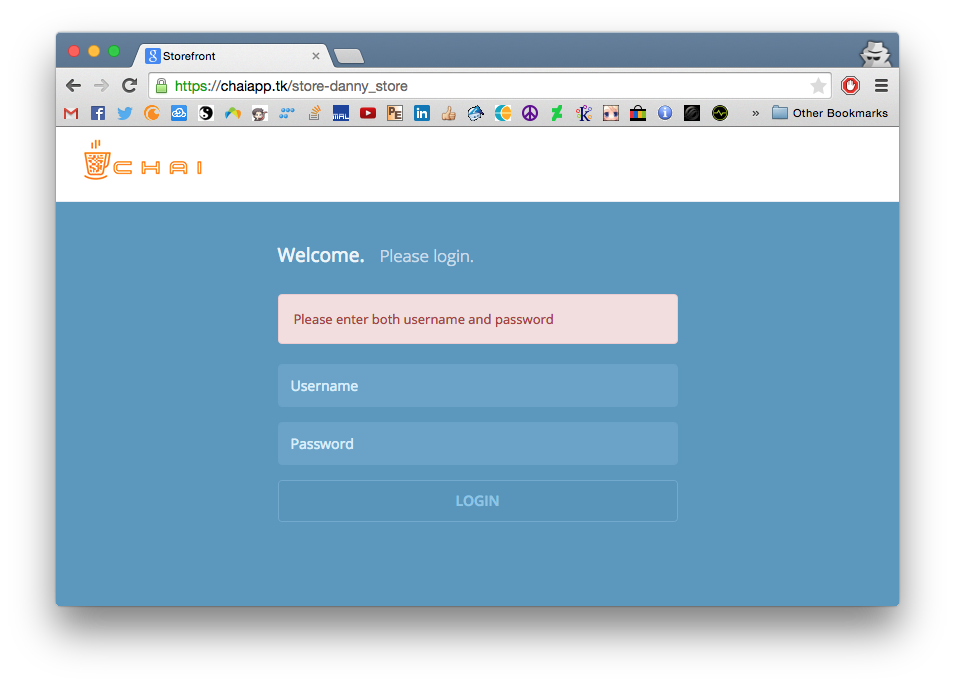
event.preventDefault() // Do no submit

};

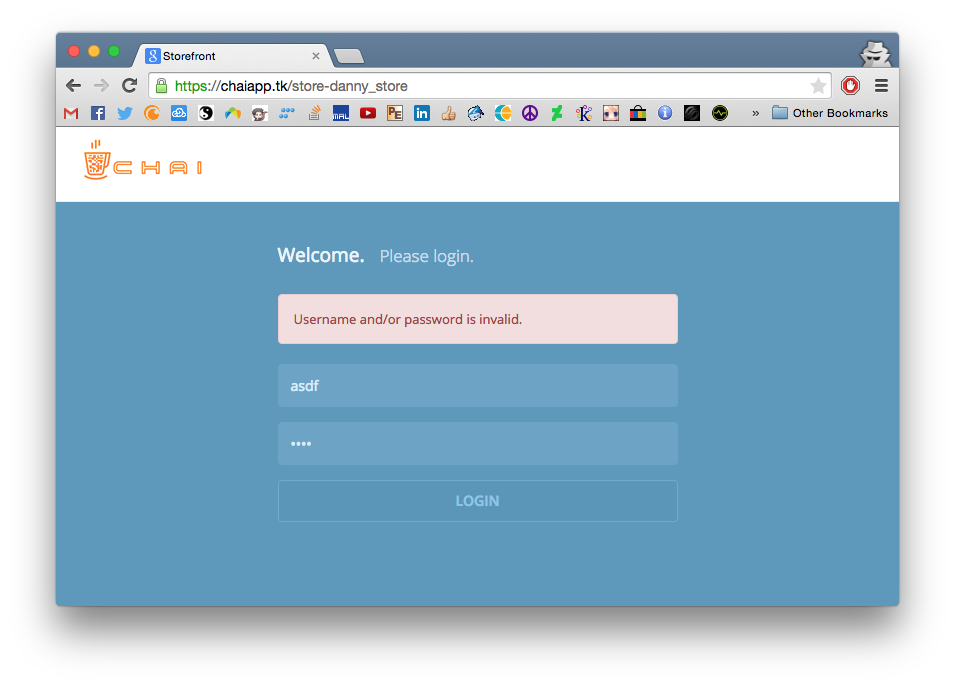
});

**Exceptions:**

Username and/or password missing (caught by JavaScript):



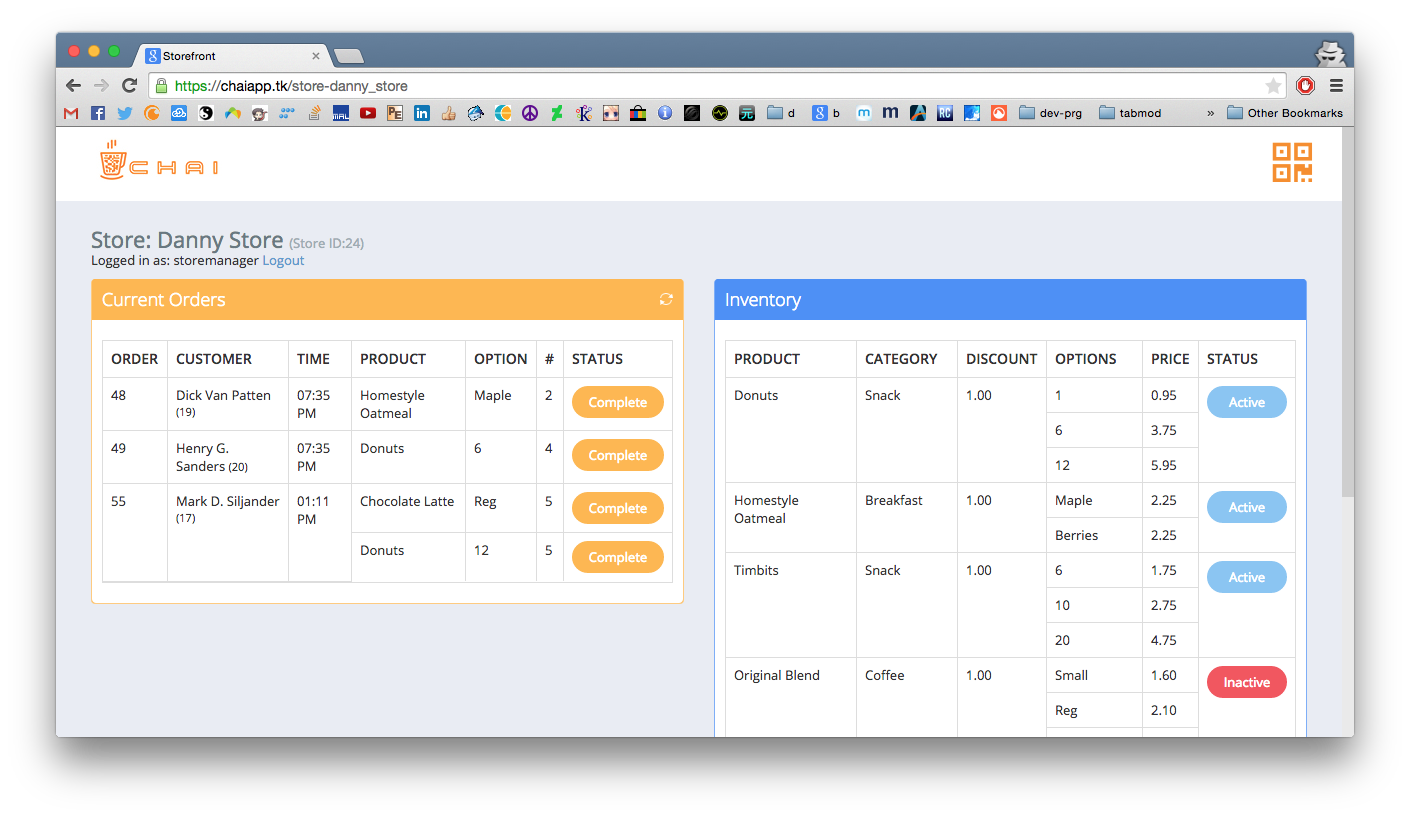
Username and/or password incorrect (caught by cloud, renders page with alert text parameters):



**Successful Login:**

Once logged in, the store landing page will be redirected. The username will appear under the site header text along with a Logout button.

### Store Landing Page



**CSS themes used:**

Metronic v3.6.1 (<http://keenthemes.com/>)

Flat UI (<http://designmodo.github.io/Flat-UI/>)

**Django variables:**

|  |  |
| --- | --- |
| inventory | (List of database objects) Menu items in inventory for the store |
| orders | (List of database objects) All orders for the store |
| WS\_URL | (String) Websocket server URL |
| current\_orders | (List of database objects) Current orders for the store |
| store\_name | (String) store short name (format “store\_name”) |
| store\_id | (String) store ID |

**Key Elements:**

|  |  |
| --- | --- |
| Current orders: Complete button | Form submit button, parent form id=“complete\_order” |
| Inventory: Active/Inactive button | Form submit button, parent form id=“toggle\_active\_button” |
| QR code link | Form submit button, parent form class=“login-form” |
| Logout link | Link, id=“logout-form” |

**Events pseudocode: Websocket listening** (~/chai\_cloud\_deploy/templates/api/stores.html Lines 345-350)

// Define a new websocket

window.socket = {}

// Listen to our WS server address

socket.ws = $.gracefulWebSocket("{{ WS\_URL }}")

socket.send = function (message) {

socket.ws.send(message)

}

// Define event on message recieved from WS server

socket.ws.onmessage = function (event) {

// Gets and parse a WS request into JSON

var messageFromServer = event.data

parsed = JSON.parse(messageFromServer)

// Do some stuff with the messages...

};

**Events pseudocode: Listen for new order** (~/chai\_cloud\_deploy/templates/api/stores.html Lines 353-477)

// Got message from websocket server

...

if (parsed['action'] == "order") {

//New customer order

order\_info = parsed['order\_info']

// Parse into HTML

html = build\_html(order\_info)

// Append to our

append\_to\_order\_table(html)

// Add event listener to the new button

...

}

};

**Events pseudocode: Add event listeners to buttons** (~/chai\_cloud\_deploy/templates/api/stores.html Lines 478-571)

// Define buttons

completeButton = $("#complete\_order")

activeButton = $("#active\_button")

// Inventory button events

for (i=0; i<activeButton.length; i++) {

// Add click event

activeButton[i].addEventListener('click', function(event) {

// Change some html elements (such as button color after click)

change\_some\_html()

// POST request to inventory active/inactive API

xmlhttprequest("store-{{ store\_name }}/" + params)

})

}

// Complete button events

for (i=0; i<completeButton.length; i++) {

// Add click event

activeButton[i].addEventListener('click', function(event) {

// Change some html elements (such as row visibility after click)

change\_some\_html()

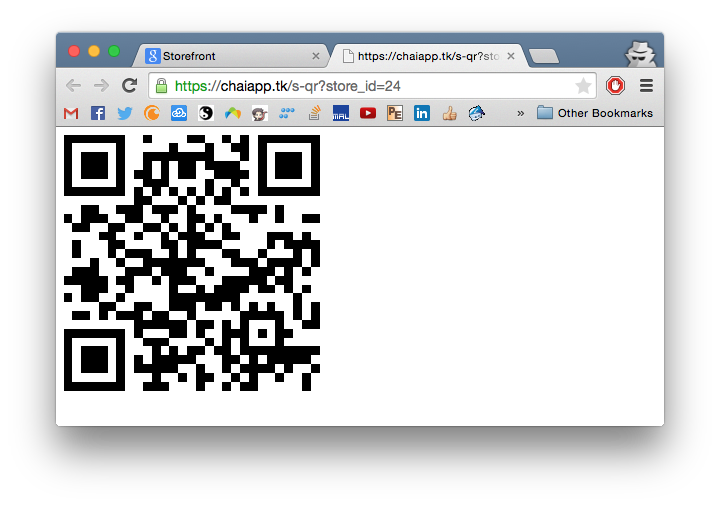
// POST request to order complete update API

xmlhttprequest("store-{{ store\_name }}/" + params)

})

}

### QR code page



The QR code link on the store landing page redirects to this QR code generation page. This page gets a JSON string from the redirect from our web service and generates a QR code representation of the string. The default JSON string only includes the store ID of the previous page before redirection. This is sufficient for the customer app to scan to check-in, but is flexible to customization. The generated QR code is an image file, can easily be put into any graphics template.

**Django variables:** json: (String) JSON string used to generate QR code

**Javascript Pseudocode:** (~/chai\_cloud\_deploy/templates/api/qr.html Lines 15-21)

// Use QR generator library to generate new QR code

qrcode = new QRCode()

// Make QR code of JSON located in hidden input with id="data"

qrcode.makeCode($("#data"))

## Front-end: Customer App

### Development Environment

The customer app is written in Monaca.io, a Cordova-based26 online IDE and compiler for cross platform app development. Development is HTML5 based, allowing interoperability with any platform with Webview functionality. This includes Android27, iOS29, Windows 8 and Chrome Apps.

Because it is an online compiler, compiler settings and results are uniform across the different development environments. As long as the source code is the same, it will compile the same way regardless of the underlying system. As it is HTML5 based, only HTML code and configuration files are required to import a Monaca.io project. Our project source is located: <https://github.com/dannyyiu/410coffee/tree/master/src/chai_app>

Our project was optimized for iOS, but has been tested on both iOS and Android to be fully functional.

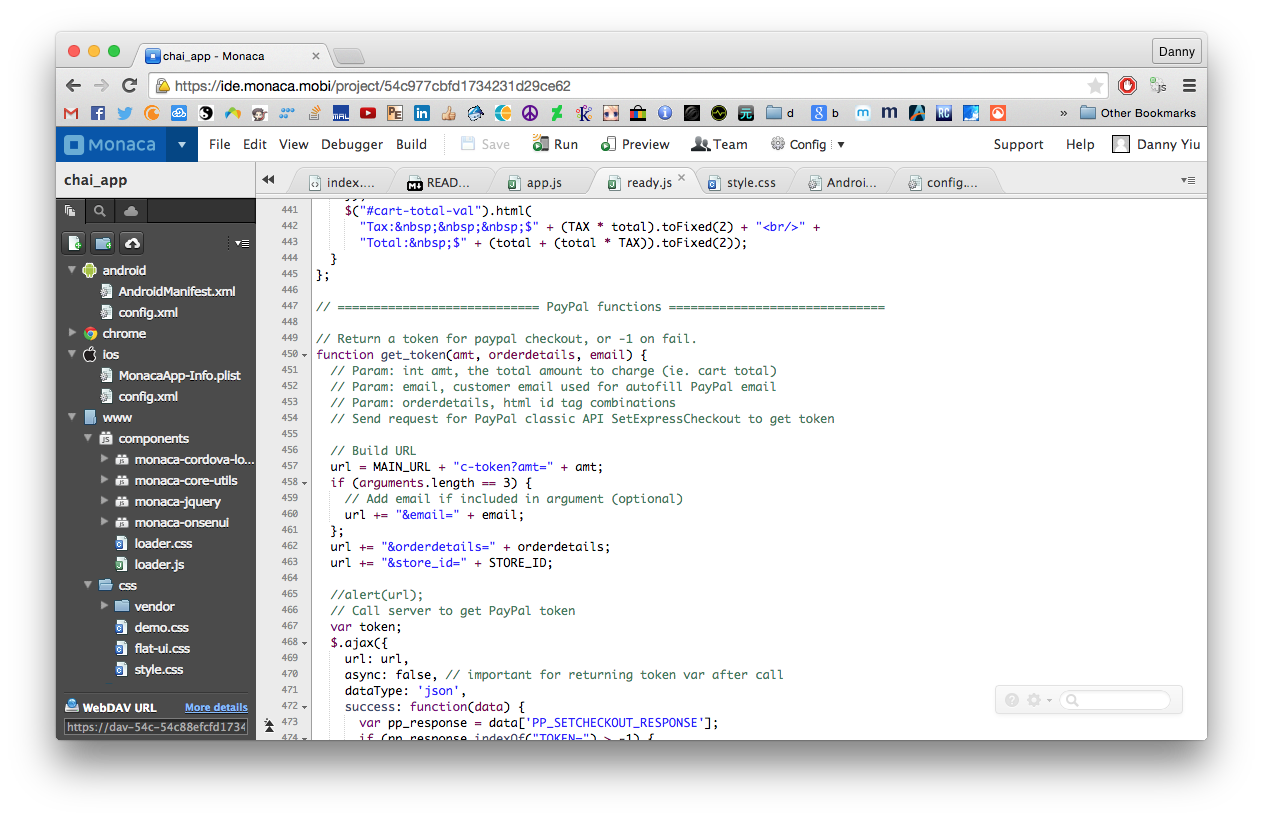


Figure : Monaca.io IDE

**Monaca.io project structure**

|  |  |
| --- | --- |
| /www/index.html | Main HTML source file |
| /www/js/app.js | Page load init event scripts |
| /www/js/ready.js | Page ready event scripts |
| /www/img | App images |
| /www/css | CSS files |
| /config.android.xml | Android configuration file |
| /config.ios.xml | iOS configuration file |

**Cordova version:** 4.1.0

**CSS theme used:** Onsen UI (<http://onsen.io/guide/overview.html>)

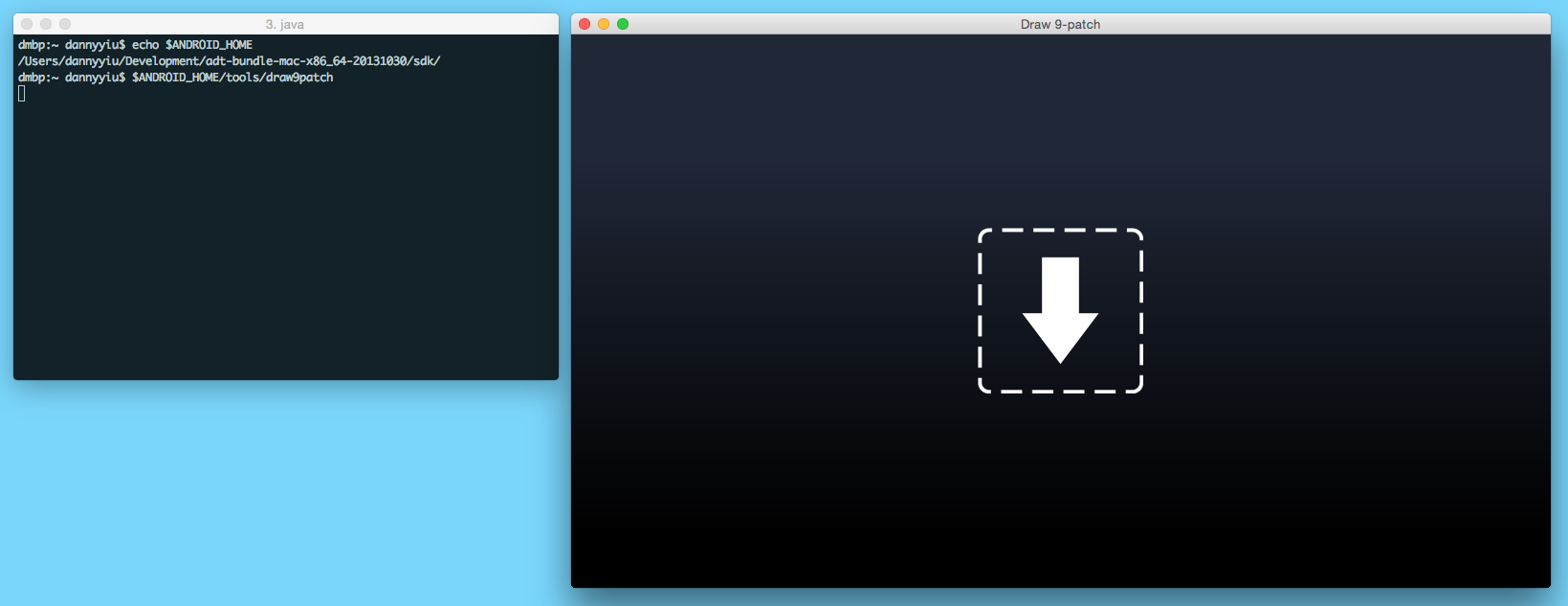
**Cordova Plugins:**

* InAppBrowser: <http://docs.monaca.mobi/2.9/en/reference/phonegap_29/en/inappbrowser/inappbrowser/>
* Splashscreen: <http://docs.monaca.mobi/2.9/en/reference/phonegap_29/en/splashscreen/splashscreen/>
* BarcodeScanner: <http://docs.monaca.mobi/cur/en/reference/phonegap_plugin/barcode_scanner/>
* Monaca Plugin (core features, required for Monaca.io)

#### Android Specific Notes

The splash screen for Android requires a special format called 9-patch29 PNG. Since screen sizes for Android vary vastly unlike iOS, the splash screen is defined by a set of images with a 1-pixel edge marking parts to stretch and parts to maintain ratio.

To create 9-patch PNG, Android SDK comes with a tool: $ANDROID\_SDK/tools/draw9patch



9-patch has two things to define: 1) What to stretch and 2) what to maintain size. Stretch areas are defined on the left and top edges, while size maintain is defined on the right and bottom edges. A 1-pixel black line is drawn on those edges for areas defined:

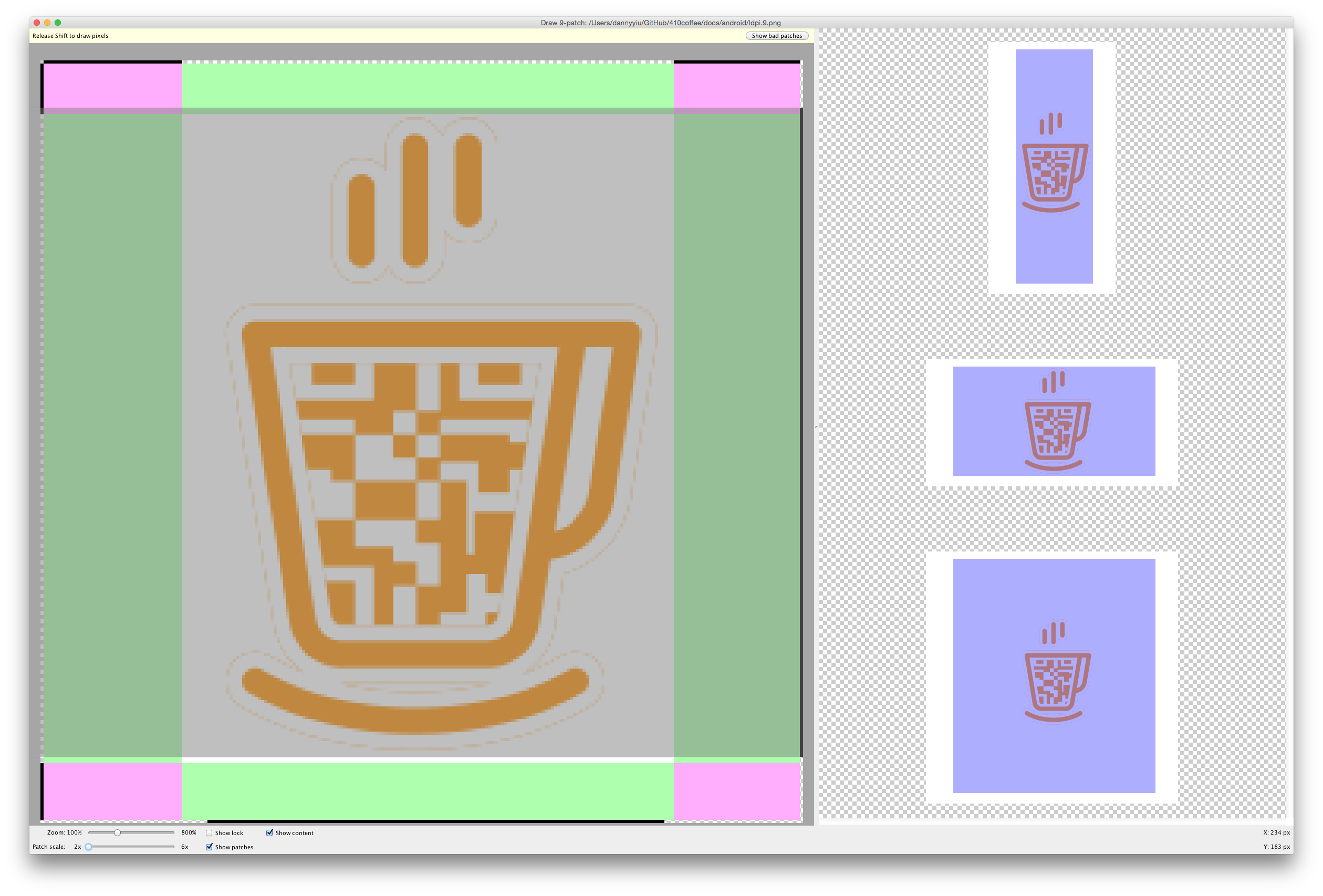


Figure : Android draw9patch tool

#### iOS Specific Notes

To build iOS apps, even debug versions, the following are required:

* Development certificate file for iOS
* Device UDIDs30 for development
* Provisional profile file

The above can all be registered and downloaded with at the Apple Developer website: <https://developer.apple.com/account/ios/overview.action>

### App Flow

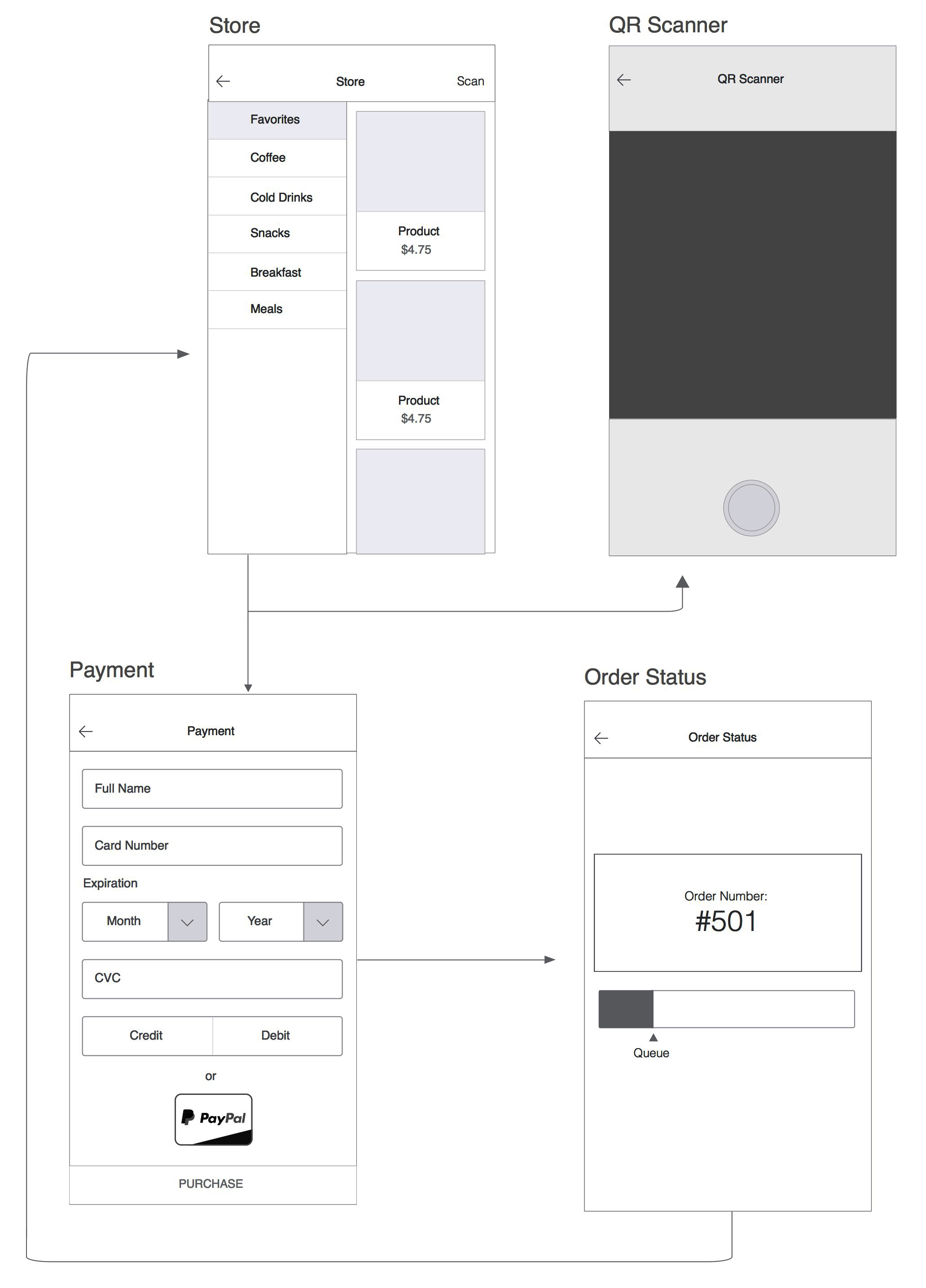


Figure : Logic flow of the customer app. Each page will be discussed in detail below.

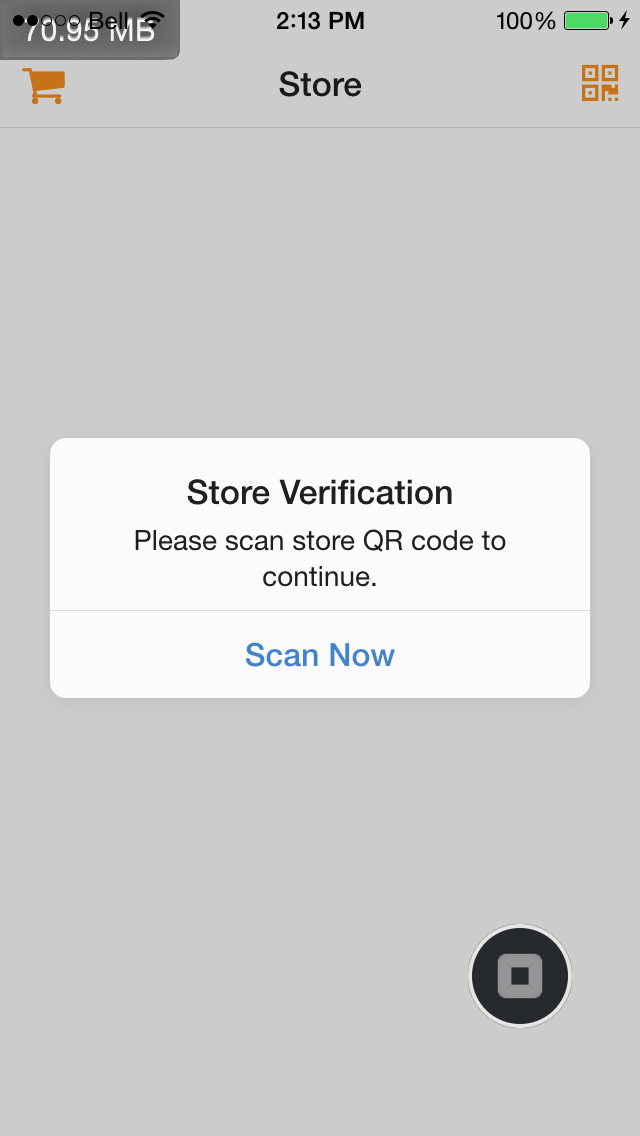
### 

### Store Page: Before QR Code Scan

For the app to retrieve the menu for a store, it must first know which store it is in. For this to happen, it must first scan a QR code. Customers opening the app will be required to scan a QR code before continuing.

**Button events:**

|  |  |
| --- | --- |
| Scan Now | Call BarcodeScanner plugin, open camera page for QR code scanning |



**Page pseudocode:** (/www/js/app.js Lines 68-80)

// Check if not scanned yet

if (!('store\_id' in options)) {

// Onsen UI alert object

ons.alert({

message: Please scan store QR code to continue.',

title: 'Store Verification',

buttonLabel: 'Scan Now',

callback: function() {

// Go to scan page on button press event

scan()

}

})

}

### 

### QR Scanner

When the scan function is called, it initiates the BarcodeScanner plugin and opens a new view for the camera to scan the QR code. This is where the customer scans the store QR code. Success returns a result object, scan error triggers an error string.



**Page pseudocode:** (/www/js/ready.js Lines 77-98)

// Check if not scanned yet

function scan() {

// Initialize BarcodeScanner plugin

barcodeScanner.scan(function(result) {

if (‘store\_id’ in result) {

// Redirect to Store page with store ID parameter

resetToPage( store\_page, result[‘store\_id’] )

}

}, function (error) {

// Scan error, alert error message

alert(“Scan error”)

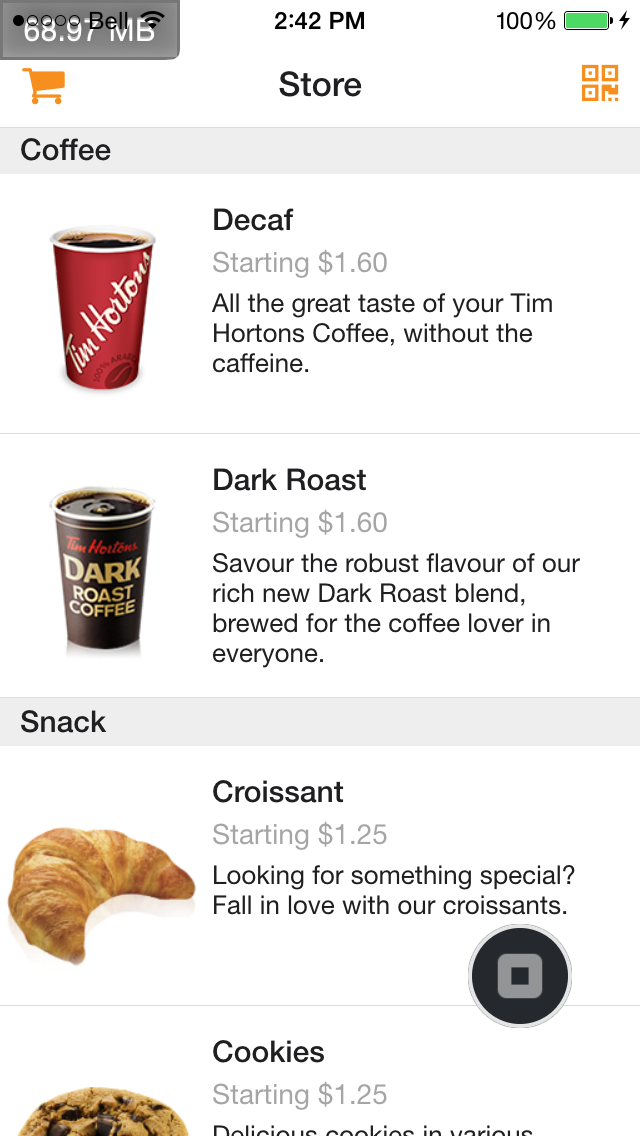
})

}

### 

### Store Page: Menu

Once a customer has scanned the QR code, the customer is returned to the store page with a store ID. This is then used to retrieve the store menu information using our customer API c-menu. Only active inventory items will be displayed. All menu information will be displayed by their categories.



**Request API:** c-menu/

**Request parameters:**

store\_id: (String) Store ID

**Response JSON format:**

{ category\_name :

{ prod\_name :

{ 'img\_url' : img\_url,

'prod\_id' : prod\_id,

'prod\_name' : prod\_name,

}, ...

}, ...

}

**Item id tag format:**

prodid-<product id>

**Page pseudocode:** (/www/app.js, Lines 14-67)

// Check if not scanned yet

if ('store\_id' in options) {

// Call c-menu API

$.getJSON(MAIN\_URL + 'c-menu?store\_id=' + options.store\_id,

function(data){

// Get JSON string response

response = json\_string(data)

// Build HTML string to append to menu

// Add click event to each list item

html += "<ons-list-item>" + .... + data + ...

// Append HTML

content.innerHTML = html

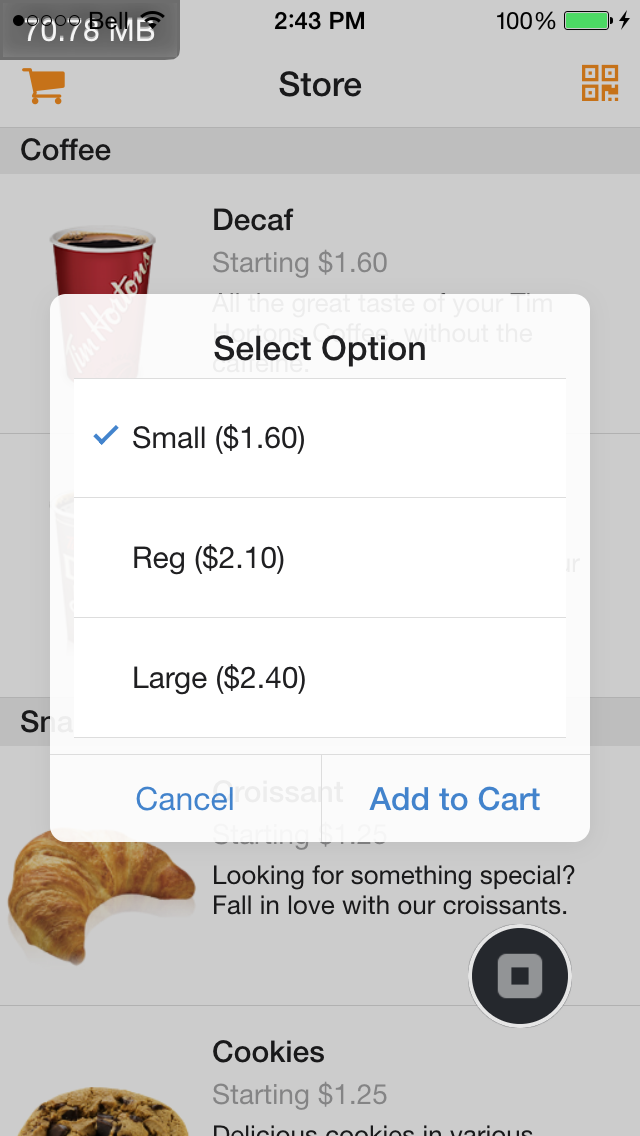
})

}

### 

### Store Page: Options

Each item can have set of options (for example, coffees can have regular and large sizes), and each option might have its own price difference from the base price. When an item is clicked, an event is triggered to request option information from our customer API c-prod using the product ID of that item.



**Request API:** c-prod/

**Request parameters:**

prod\_id: (String) Product ID

**Response JSON format:**

{ op\_id :

{ op\_name : price }, ...

}

**Item id tag format:**

opid-<option id>

**Button events:**

|  |  |
| --- | --- |
| Cancel | Return to store page |
| Add to Cart | Add to cart and show cart dialog |

**Page pseudocode:** (/www/ready.js, Lines 103-212)

// Call API

$.getJSON(MAIN\_URL + "c-prod?prod\_id=" + prod\_id,

function(data) {

// Build HTML string for option dialog

html += "<ons-list-item>" + ... + data + ...

})

// Put html in a dialog

ons.dialog({

innerHTML: html,

title: "Select Option",

buttonLabels: ['Cancel', 'Add to Cart'],

callback: function(index) {

if (index=1) { // Add to Cart button pressed

// Add product and its option to cart

cart\_add(prod\_id, option\_id)

// Hide item in menu (since it’s already added to cart)

menu.hide(“#opid-“ + op\_id)

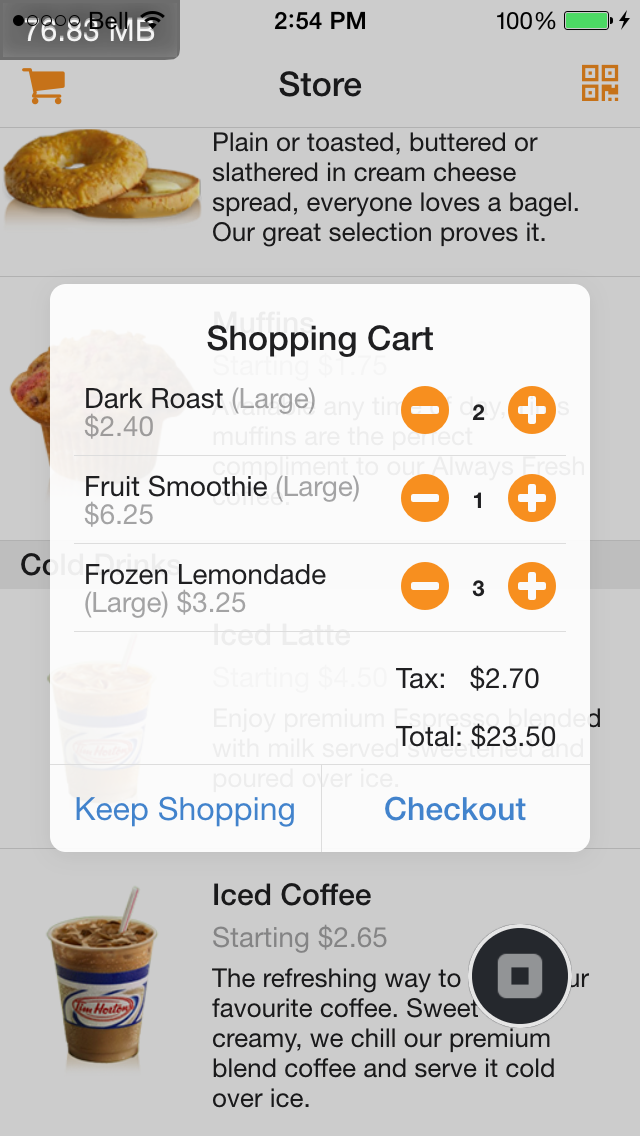
}

}

})

### Store Page: Shopping Cart

The shopping cart contains a record of all items and their options, as well as a calculated total price. There are cart update buttons to add or remove quantity to each cart item, which also updates the total automatically. When quantity reaches 0, the item is removed from the cart, and reappears in the store menu. Checkout button calls the /



**Item id tag format:**

cart-<product id>-<option id>

**Button events:**

|  |  |
| --- | --- |
| Keep Shopping | Close cart dialog |
| Checkout | Calls c-token API and proceed to proceed to PayPal In-context Express Checkout page with the token response. |

**Page pseudocode:** (/www/ready.js, Lines 235-273)

// Update cart total price on show

cart\_total\_update()

// Get cart HTML

html = get\_cart\_html()

// Put html in a dialog

ons.dialog({

innerHTML: html,

title: "Shopping Cart",

buttonLabels: ['Keep Shopping', 'Checkout'],

callback: function(index) {

if (index=1) { // Add to Cart button pressed

// Call c-token API

get\_token(order\_list, payment\_total)

// Use token for PayPal In-context page

// Open in InAppBrowser plugin

window.open(PAYPAY\_URL + token)

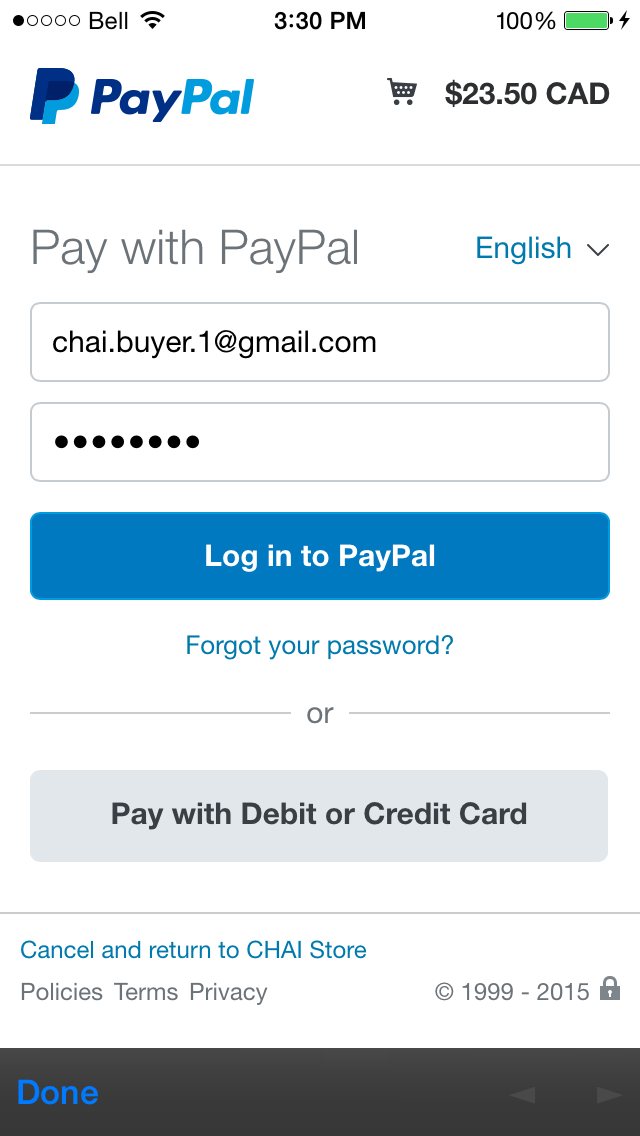
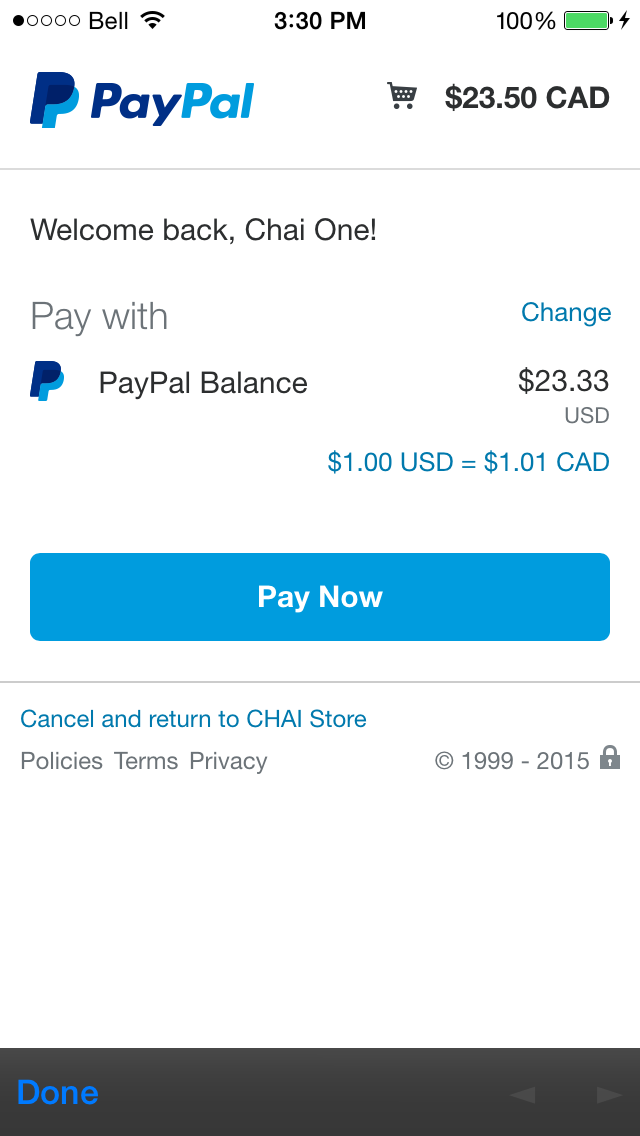
}

}

})

### Payment: PayPal

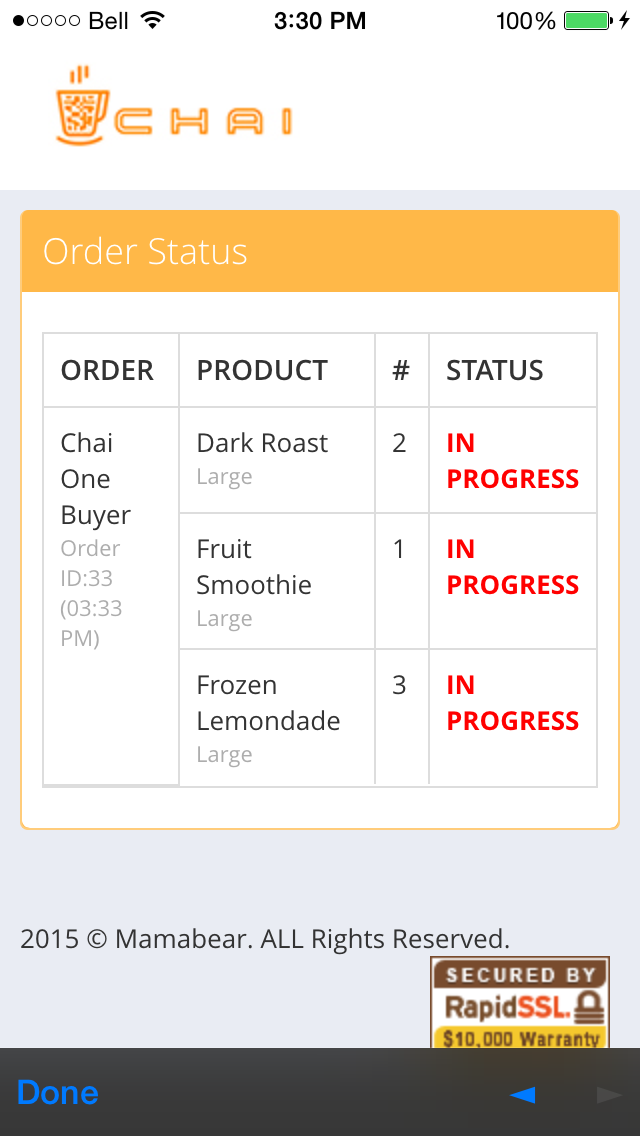
After checkout is pressed, the InAppBrowser plugin shows a window that redirects to PayPal’s In-context Express Checkout page with the token received from the previous call to the c-token API. From here on, PayPal handles all customer data. The user is prompted to log in, or choose to pay directly with credit card. Once logged in (or confirmed credit card information), the user will be redirected to a Pay Now page.

### 

### Order Status Page

Once the “Pay Now” button is pressed, the customer will be redirected to customer order status page with the API c-order-status-<store name>, with the token returned from PayPal



**Request API:** c-order-status-<store name>/

**Request parameters:**

‘token’ : String, PayPal token

‘PayerID’ : String, PayPal payer ID.

**Response template:**

~/chai\_cloud\_deploy/templates/api/order-status.html

**CSS themes used:**

Metronic v3.6.1 (<http://keenthemes.com/>)

**Source files:**

~/chai\_cloud\_deploy/templates/api/order-status.html

~/chai\_cloud\_deploy/api/views.py (Lines 391-474)

**Django variables:**

|  |  |
| --- | --- |
| customer\_orders | (List of database objects) Active orders by this customer |
| store\_name | (String) Store short name (format “store\_name”) |
| WS\_URL | (String) Websocket server URL |

**Key Elements:**

|  |  |
| --- | --- |
| Status per item | <td> tag, id=“det-<order detail ID>” |

**Events pseudocode: Websocket listening** (~/chai\_cloud\_deploy/templates/api/order-status.html Lines 28-45)

// Listening to websocket

...

// Message recieved event

socket.onmessage = function(event) {

// Get JSON of message recieved

message = json(event.data)

// Check if message contains order completion

if (message['complete'] == 'detail\_complete') {

// If order completion is for customer's order detail ID...

$("#det-" + detailid).text("COMPLETE");

$("#det-" + detailid).css("color", "green");

// Note this will be ignored if the ID is not the customer's

}

}

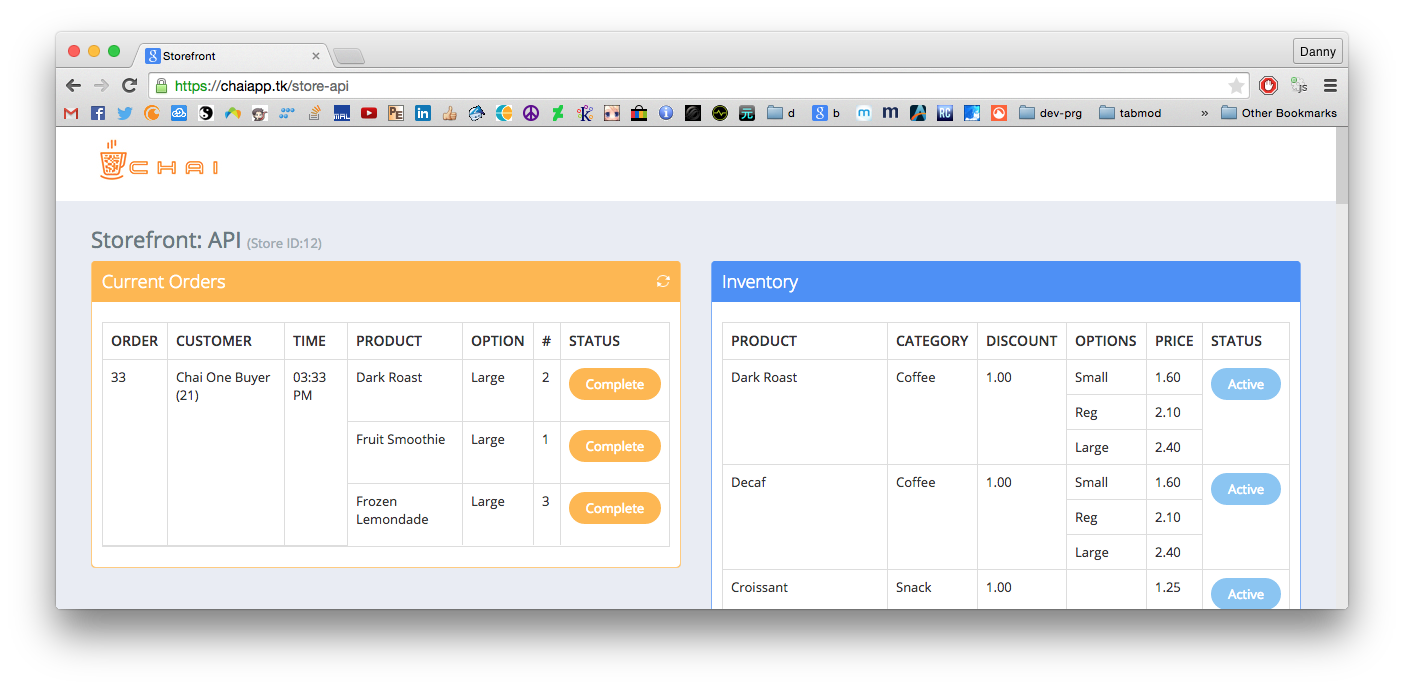


Figure : Orders will show up on the store landing page in real-time

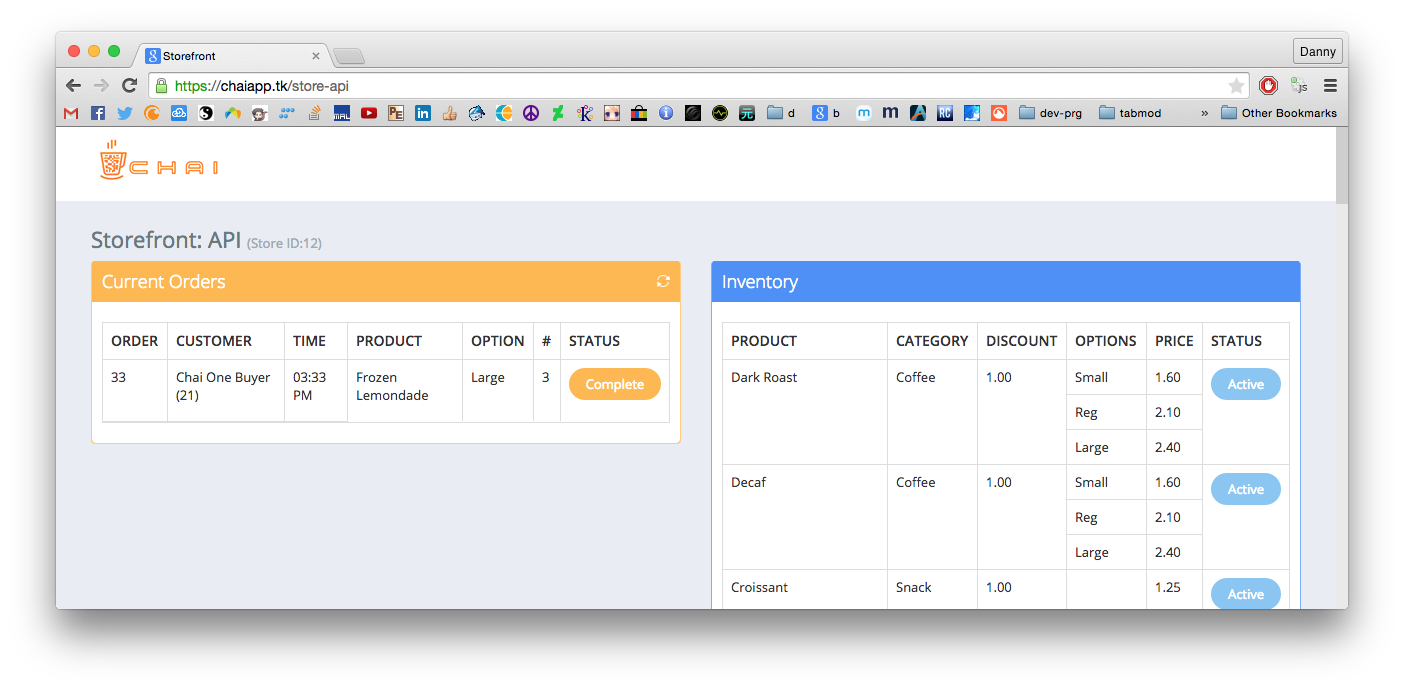


Figure : Store employees can mark the orders as complete by clicking on the "Complete" button

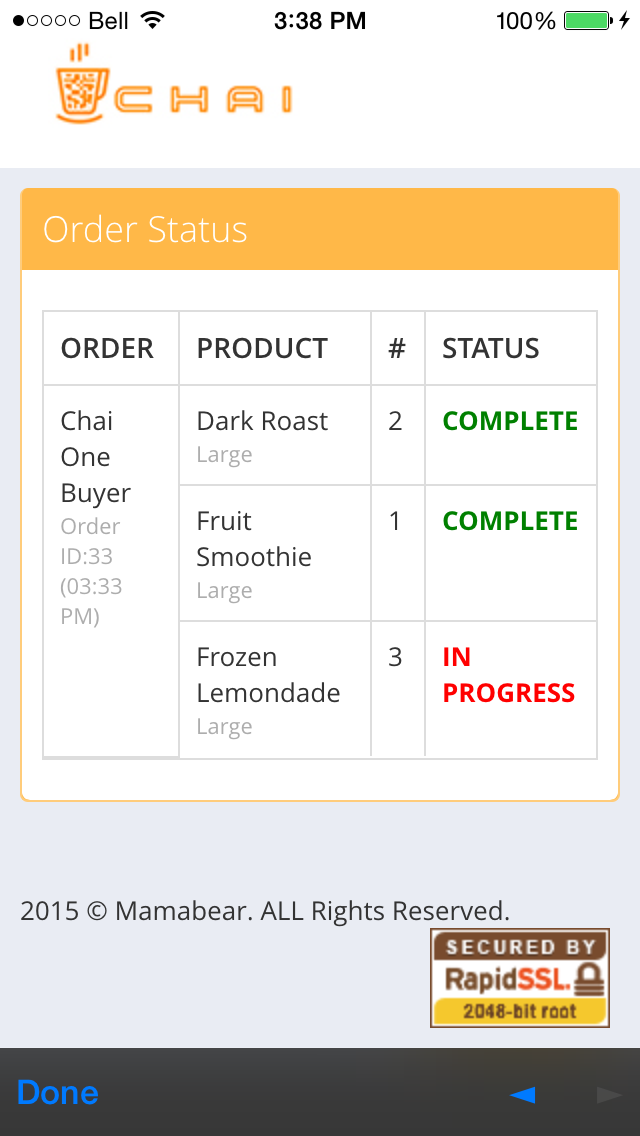


Figure : Without refreshing, the customer will see order completion in real-time on their app

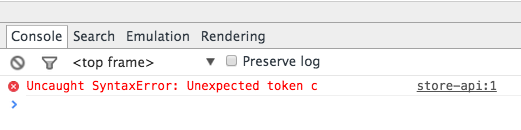
## 

## Known Issues and Future Improvements

### Admin Page

* The current template uses Django’s admin template. Although there are no security risks, aesthetics are dull. Improvements can include integrating a CSS theme into the admin templates.
* There is no image uploading functionality yet. All images have to be manually uploaded through SFTP. Although this is only required for the first time (and updates), this can be a convenient improvement.

### Store Landing Page

* Slight table width problem when window width is between 992px and 1156px, and also when under 565px. This is a CSS issue, can be fixed either by reducing table columns or altering the CSS files involved. Other width dimensions are not affected.
* Strange recurring “Uncaught SyntaxError: Unexpected token c” on page load JavaScript console log. However, this has no effect on functionality:  
  
* Some CSS styling is still hardcoded into the HTML file. This can be cleaned up by putting them in a separate CSS file.
* Tables are currently used for the store landing page. For rows with sub-rows, this gets messy. Migrating to div and span may be a better choice.
* Both JavaScript and jQuery are used together. This is fine in most places, but some functions can be unified for cleaner code (such as JavaScript XmlHttpRequest functions and jQuery $.ajax can be unified).

### Customer App

* Order status page for the Android app does not load mobile site CSS. This is perhaps an incorrect User-agent header used by the InAppBrowser plugin for Android. One possible solution is to force the mobile site CSS on the ~/chai\_cloud\_deploy/templates/api/order-status.html template.
* Currently only one option can be selected per item. Future improvement can be adding support for multiple options selected per item (for example, being able to order two different sizes of coffees)
* Since the shopping cart is currently a pop-up, the amount of items is limited by the phone’s screen height (roughly 10 items). This can be improved by making the dialog scrollable or separating the shopping cart into its own page.
* Option selection dialog list items have a solid white background. This should be changed to slight opacity like the rest of the dialog. Can be done through CSS.

### Server

* Current Apache SSL settings only allow hard typed SSL URLs. A future improvement can be to redirect all HTTP requests to HTTPS. This is optional though, since all cloud access are done either by internal staff or through API calls programmatically.
* Port 1025 is used for the websocket server, but this can be integrated into port 443 by reverse proxy settings in Apache.
* All data is stored in the sqlite file db.sqlite3. This file can be copied for backup. A script can be written for this.

# Glossary

1. Amazon EC2: Scalable cloud computing service. <http://aws.amazon.com/ec2/>
2. Python: Object-oriented scripting language. <https://www.python.org/download/releases/2.7/>
3. Sqlite3: File based database. <https://docs.python.org/2/library/sqlite3.html>
4. Ubuntu: Debian-based Linux distribution. <http://www.ubuntu.com/>
5. SSH: Secure shell.
6. t2.micro: Amazon micro instance. <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-instances.html>
7. Elastic IP: Static IP address for dynamic cloud computing. <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html>
8. Websocket: Full duplex TCP connection. <http://www.html5rocks.com/en/tutorials/websockets/basics/>
9. EC2 key pair: public and private key used for encryption. Private key is saved as a .pem file. <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html>
10. DNS: Domain name system
11. PuTTY: SSH client. <http://www.chiark.greenend.org.uk/~sgtatham/putty/>
12. Intermediate CA bundle: Intermediate ertificate authority, used to establish chain of trust with the root CA. <https://support.globalsign.com/customer/portal/articles/1224583-root-intermediate-certificate-bundles>
13. PayPal Express Checkout: PayPal payment system. <https://developer.paypal.com/docs/classic/express-checkout/integration-guide/ECGettingStarted/>
14. PayPal Classic API: Methods called to PayPal’s webservice. <https://developer.paypal.com/docs/classic/api/>
15. Regular Expressions: Strings used as representations of search patterns. <http://www.regular-expressions.info/>
16. MVC: Model-view-controller, service framework architecture.
17. API: Application Program Interface. <http://www.webopedia.com/TERM/A/API.html>
18. REST: Representational State Transfer. <http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html>
19. AJAX: Asynchronous JavaScript and XML. <http://www.w3schools.com/ajax/>
20. JSON: JavaScript Object Notation. <http://json.org/>
21. SetExpressCheckout: PayPal Classic API method. <https://developer.paypal.com/docs/classic/api/merchant/SetExpressCheckout_API_Operation_NVP/>
22. GetExpressCheckoutDetails: PayPal Classic API method. <https://developer.paypal.com/docs/classic/api/merchant/GetExpressCheckoutDetails_API_Operation_NVP/>
23. DoExpressCheckoutPayment: PayPal Classic API method. <https://developer.paypal.com/docs/classic/api/merchant/DoExpressCheckoutPayment_API_Operation_NVP/>
24. NVP: Name-Value Pair, URL parameter format. <https://developer.paypal.com/docs/classic/api/NVPAPIOverview/>
25. QR code: Quick Response Code, portable barcode type.
26. Cordova: A set of device APIs used for hardware access with Javascript. <https://cordova.apache.org/>
27. Android: Mobile operating system by Google. <https://www.android.com/>
28. iOS: Mobile operating system by Apple. <https://www.apple.com/ca/ios/>
29. 9-patch: An image format defining areas of stretch, used by Android. <http://developer.android.com/tools/help/draw9patch.html>
30. UUID: Universally Unique Identifier.

# References

Wester, J. (2012). Starbucks named in patent infringement suite over mobile payment app. *Mobile Payments Today*. Retrieved from: <http://www.mobilepaymentstoday.com/articles/starbucks-named-in-patent-infringement-suit-over-mobile-payment-app/>

Ankeny, J. (2013). Starbucks mobile payments eclipse 10% of U.S. store transactions. *FierceMobileIT*. Retrieved from: <http://www.fiercemobileit.com/story/starbucks-mobile-payments-eclipse-10-us-store-transactions/2013-07-29>

Marko, K. (2013). The business case for going mobile. *Information Week: Network Computing.* Retrieved from: <http://www.networkcomputing.com/wireless-infrastructure/the-business-case-for-going-mobile/a/d-id/1234348>?