Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

CheckedOut– Design Document

28th November, 2016

Introduction

Our group plans to create a user-friendly Android and web-based application that allows users to create shopping lists and plan shopping routes based on prices and/or proximity of stores. It will also provide user profiles and recommend items to the users. This document will explain the architecture and design of CheckedOut.

System Architecture and Rationale

CheckedOut's design consists of the following major components:

1. Main Server that handles queries to either store or retrieve information from Main Database.
    - This server is connected to a MongoDB database on the same virtual machine and handles JSON formatted requests from the web and Android clients. It provides the processing for many features of the product and relays this information back to the clients.

2. Web Crawler that crawls retailer websites to extract relevant information.
    - The web crawler runs on the main server; however, it is a major component as it populates the database for which our product relies on. It does this by relaying information gathered on the HTML pages of websites back to the main server. Furthermore, we will be following the robots.txt of each retailer to make sure we abide by the rules they have set and do not disrupt their business.

3. Web Server that runs the website, handles user requests, and communicates with the main server.
    - The web server will receive requests from the browser (when the user interacts with the website); based on the requests, the web server will update the website accordingly, sending requests to the main server to retrieve data when necessary.

4. Android application that provides a mobile version of our services.
    - The application contains all the major functionalities of the website, but displays it in a condensed and simplified manner that provides a more intuitive user experience. It can be used on all phones running Android 4.4+, and some of the functionalities such as shopping lists can be accessed offline.

Development Decisions

Android

Android is the world's most widely used mobile platform and allows for the development of applications that are convenient for "on the go" use. Development is typically done using Java in Android Studio, which allows for the use of a variety of open source APIs and Google APIs including

Google Places. We will be targeting Android versions 4.4 and up, as it allows us to use more recent features and design elements.

Python

Our team compared the usage of Java and Python as our language for the main server as they both provided means to carry out networking capabilities through standard libraries; however, we decided to use Python as it enables faster development due to its less verbose syntax and higher level of abstraction. Also, because both Python and Java could support our server, we used Python so we could learn another programming language.

TLS

In order to keep our communications secure and make sure that important information such as user profiles are not compromised, we have decided to use TLS encryption, which is an updated version of SSL encryption. Another reason why we decided to use this is because Python natively supports TLS sockets which allow for more streamlined development. One deciding factor in choosing TLS encryption over SSL encryption is that TLS is more secure and much harder to hack into. In fact, the US Government has mandated that SSL V3 will not be used for sensitive government information.[1]

Amazon Web Services

Amazon Web Services is a cloud computing platform which offers a virtual private server (VPS) which can be used to host our reverse proxy server and web server.

NGINX

NGINX is a free and open-source HTTP server. It is recognized for its fast performance, stability, and low resource consumption. We opted to use NGINX as a reverse proxy server, acting as an intermediary between client requests (from the internet) and the backend web server. Reverse proxies act as an additional layer of security and can perform SSL encryption to secure communications between browsers and web servers.

Node.js

Node.js is a free and open-source Javascript runtime environment designed for building scalable web and network applications. We chose Node.js for building our web server due to the availability of an extensive number of open-source libraries for web development (through npm).

Microsoft Azure

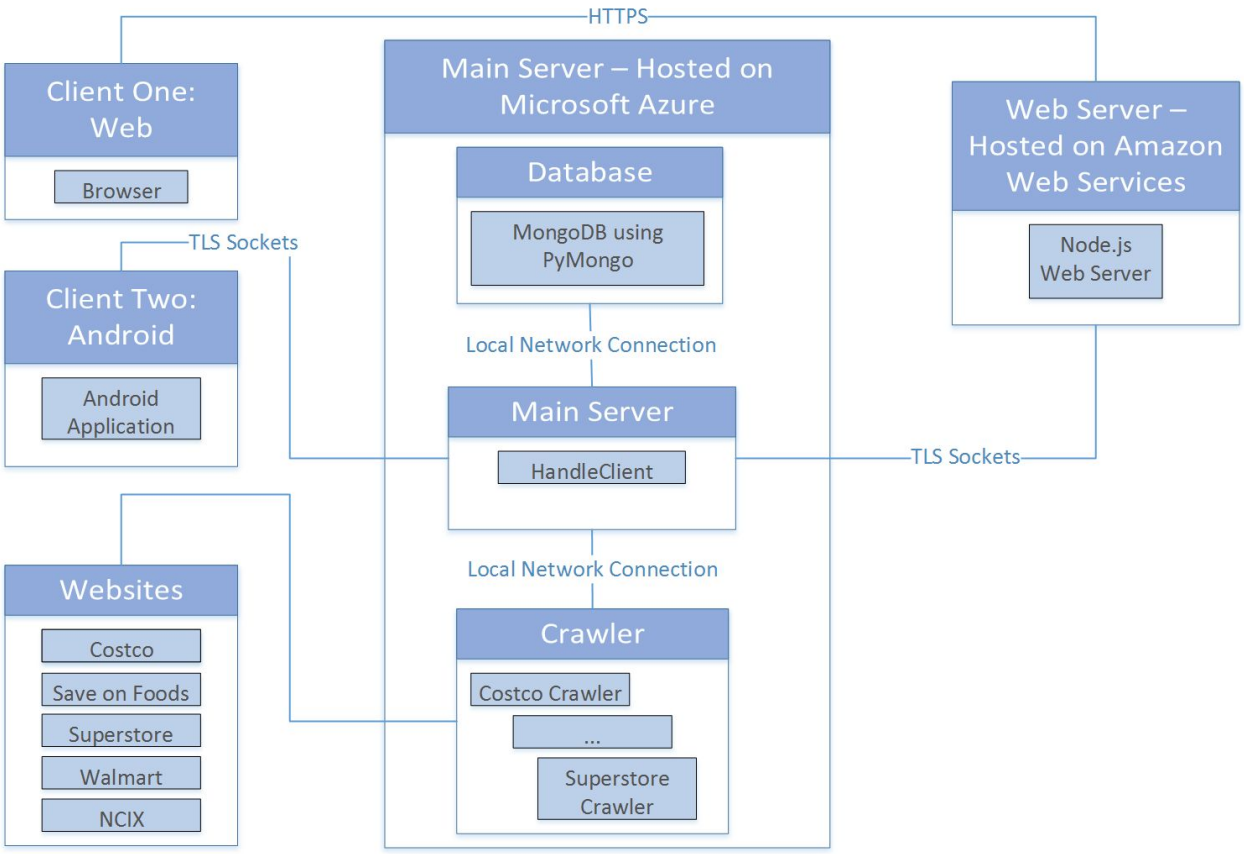
Our main python server and MongoDB database are hosted on Microsoft Azure's cloud platform. The reason why we chose to use this in addition to the AWS platform is that they had a free trial which provided a server with much better specifications (4 virtual cores, 14GB RAM) than AWS' free trial. In addition to minimizing costs, it allows us to handle requests and execute database operations much faster, which provides better performance for end users.

[1] http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r1.pdf - US Government Issued document stating its reason for moving from SSL to TLS.

Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

System Architecture

Static System Diagram

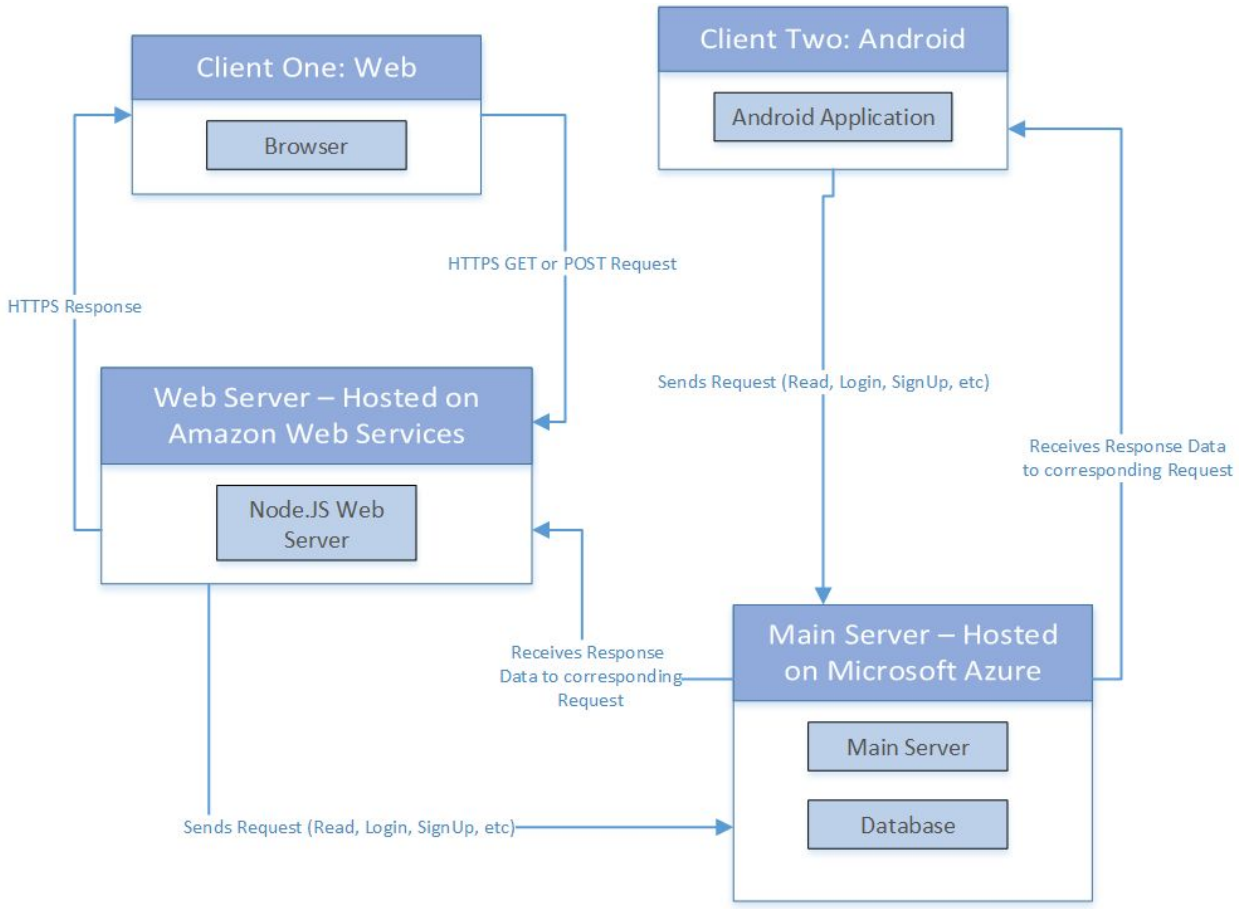


Static System Overview

Our product utilizes the client server design pattern to provide its service to users. This allows for the majority of the processing to be done on a centralized server with a local database. This eliminates the need for the client platforms to crawl webpages individually and provide product organization, which is time intensive. The website communicates with the web server using HTTPS to provide secure transmission of data, and then the web server will communicate with the main server using TLS TCP sockets, which also provides encryption. The Android app directly communicates with the main server via TLS TCP sockets. These user platforms generate requests to the main server which will query the MongoDB database, connected via a local connection. In addition, the web crawlers will read HTML pages and communicate to the main server over a local connection.

Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

Dynamic System Diagram



Dynamic System Overview

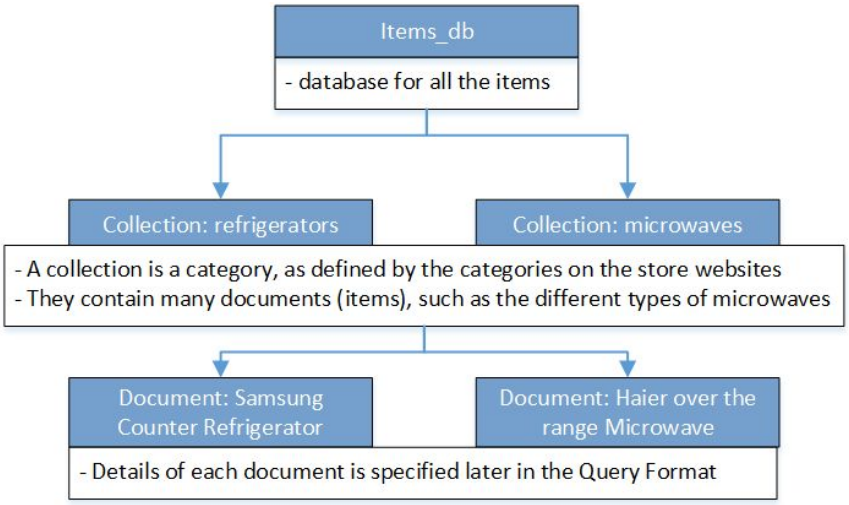
The system is structured such that the browser and the Android App are interfaces for the users to navigate the system, search for items, and create shopping lists. The Android App directly sends a request to the server while the browser requests the web server to query the main server. The main server then interprets these JSON formatted requests and queries the database for information. It then processes the data and returns the appropriate results to the client. In addition, the crawler populates the database by sending JSON formatted requests over a local connection.

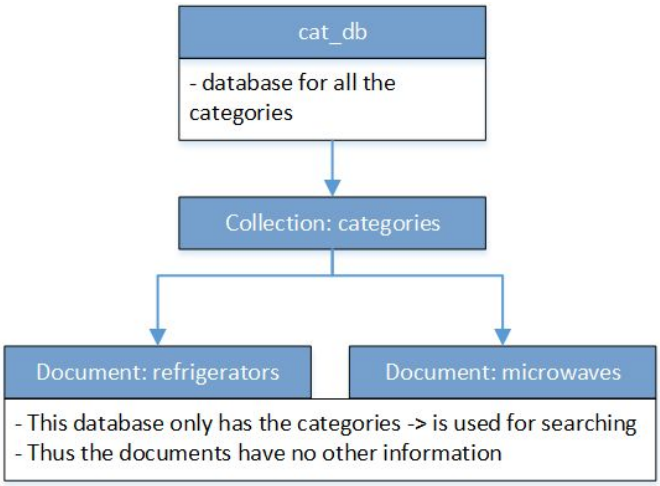
Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

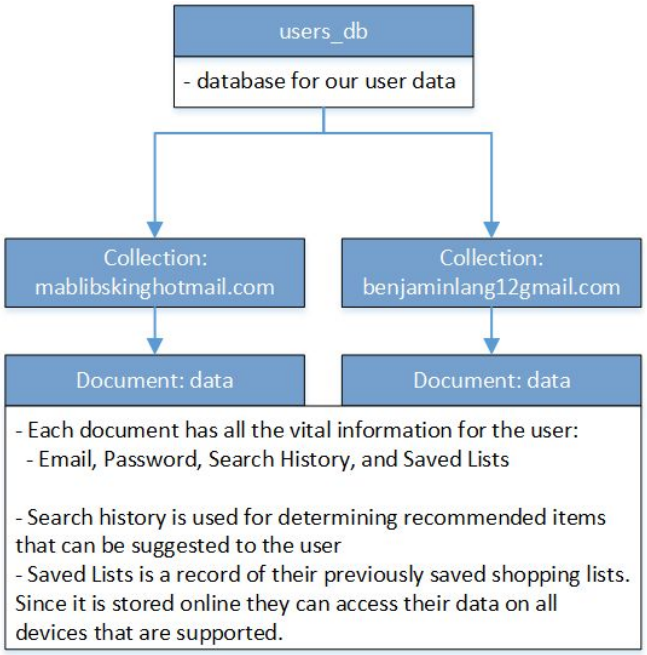
Detailed Design

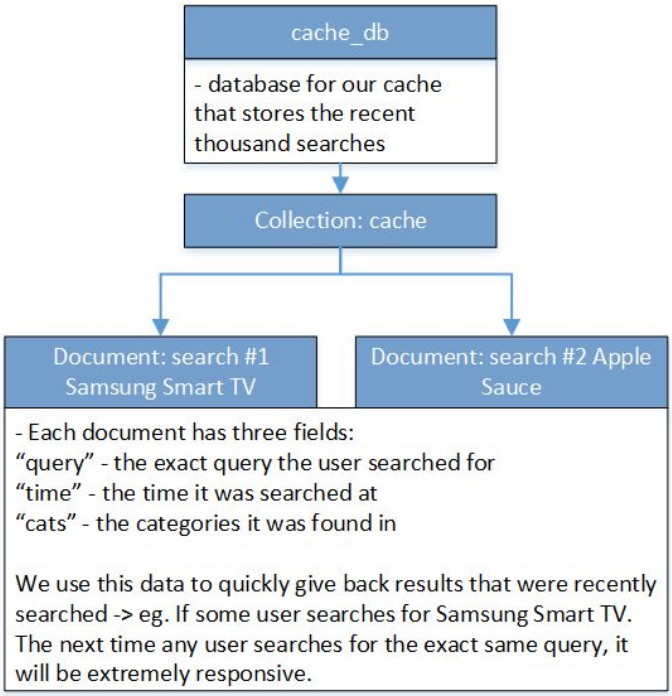
Database

Database Design Diagram









Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

MongoDB

Our team decided to use MongoDB as our database service because of the hierarchical structure that is inherently part of our data. For example, every item type will have retail items associated with it, which will then contain the associated metadata. MongoDB has a hierarchical structure built in through its database-collection-document structure, which is similar to a tree structure.

Our first choice for database services was MySQL because of its ease of use. With MySQL, it was very easy to create databases with multiple tables that can hold different data, such as Oranges and Apples. With the provided software, it was very easy to see the changes our queries made. We initially started with it and used the MySQL connector and created a working database in Python.

However, we hit a problem with scalability and merging of data. In MySQL, the data structure had to be defined before working with the database; however, in MongoDB the data structure is dynamic and can be modified to our liking at any given time. This proved to be an important as we realized with MySQL it would be very difficult to dynamically modify the tables as it requires table joining, whose performance is subpar. Because MongoDB's data structure is dynamic, we can alter data freely without worrying about all the constraints that came with MySQL.

In terms of actual database design, we have four databases: items_db, cat_db, cache_db and users_db, each storing different information. We chose to partition different databases as not only is it easier to implement and integrate into our code, but it is an active design choice we took to properly secure our data. Because we separated all of our data into different databases, it is secure against all sorts of query injection. In addition, we heavily rebuilt our query and data structure so that it takes place in multiple different steps with their implementation hidden, thus it is extremely hard for outside sources to implement query injection to extract sensitive data like user information. All details of the individual databases are as described in the above diagrams.

- items_db stores all the individual items into different categories
- cat_db stores the categories that the items are stored into
- cache_db acts as a caching system
- users_db stores all the user data

Query Structure

Our queries are structured so that every query has a message type to specify how it should be handled by the server. Keeping this in mind, the server executes different handler functions depending on the input data. Below are some examples of query and response for Read and Write (from Crawler).

Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

JSON Format Example 1:

```
- Crawler writing data into our database
- { "message_type" : "write",
    "collection" : "simcards",
    "data" : { "name" : "Samsung TecTiles, 5pk",
               "store" : "Walmart",
               "price" : "4.49",
               "url" : "store_url",
               "image" : "image_url",
               "time_updated" : "time"
              }
}

- Response from Server for a write query
- { "message_type" : "write_response",
    "status" : "success"
  }
```

JSON Format Example 2:

```
- User sends a request for Samsung
- { "message_type" : "read",
    "items" : "Samsung",
    "email" : "mablibsking@hotmail.com",
    "options" : { "num" : "2",
                  "price" : "min",
                  "store" : "Walmart"
                  }
  }

- Response from Server for a read query
- { "message_type" : "read_response",
    "items" : [
              { "data" : {"name" : "Samsung TecTiles, 5pk",
                "price" : "4.49", "store" : "Walmart", "url" :
                "store_url", "image" : "image_url"},
                "collection" : "simcards",
                "time_updated" : "time" },

              { "data" : {"name" : "Zagg invisibleSHIELD Full
                Body Protector for the Samsung
                Galaxy S II/Skyrocket",
                "price" : "8.99", "store" : "Walmart", "url" :
                "store_url", "image" : "image_url"},
                "collection" : "accessories",
                "time_updated" : "time" }
              ]
    "status" : "success"
  }
```

Example one outlines both the write query sent to the server and the write response given by the server. The write query specifies the server to write this document into the simcards collection in the database with data field containing vital information like the name of the item, the store the item is being sold at, price, and other things. The field time_updated indicates when this data was last updated in the format of: 24 Hour time – MM/DD/YY and its purpose is to provide the end-user more details about their searched items, as differing promotions may not last long. This information is crucial for our database as we plan to have automated updating of the information via the crawler, thus the database needs to know when the data was last updated.

The write response simply has a status field that signifies whether or not the write was a success, and has other possible options such as “item_insert” and “cat_insert”, signifying which of the inserts did succeed. This information is important as it allows us to properly debug and tell which operations failed.

Example two demonstrates a search in action, as we construct a read query depending on what the user has searched for. In particular, our example goes over the scenario in which the user searches for “Samsung”, sets the limit to 2 items, chooses to sort by minimum price, and sets the store to be restricted to Walmart. The email field is normally left blank unless the user is logged in, to allow us to have suggestions for the user depending on their previous searches.

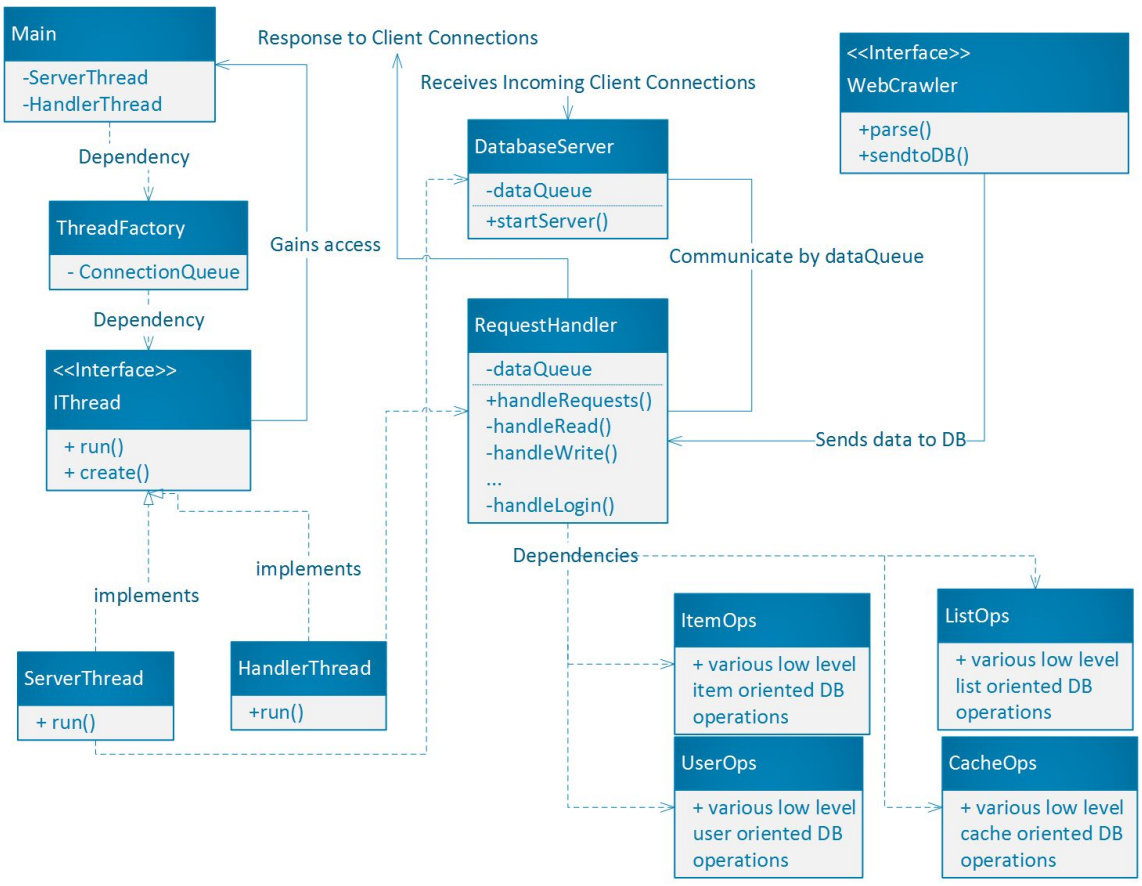
The response given back from the server contains an items field, which is an array of individual product. Each product in the items array is a result from our database, and in the case of

the example, we have specified to only return the two lowest priced products that are from Walmart. Thus, the items array only has two elements, ordered by price. The data given back is exactly as it is stored by the crawler, as we already stripped away all unnecessary data in the write operation. The data that we have also has a status field that signifies whether or not the find operation succeeded, as we catch and send "exception" in the status field if it failed. This is helpful for debugging and is used to properly integrate the applications with our server.

Main Server

The main server application will run using 4 threads to maximize the parallelism offered by our 4 virtual cores on our Azure server. One thread will be dedicated to receiving client connections while the other 3 threads will asynchronously handle the requests. This ensures that our server is nonblocking and can handle multiple separate requests at once. Additionally, having a constant amount of threads is efficient, as there is less overhead from dynamic starting and ending of threads from the OS. Upon startup, the main function will get the threads from the thread factory and run them. The threads communicate via a thread-safe blocking queue which in the server thread places connections on the queue awaiting to be handled, and the request handler threads pop off these connections to handle their query. The web crawler has multiple subtypes which crawl the different websites as each website has different HTML layouts and tags. On the Azure server, our crawlers are triggered to run every Saturday at midnight, and they will update the MongoDB database.

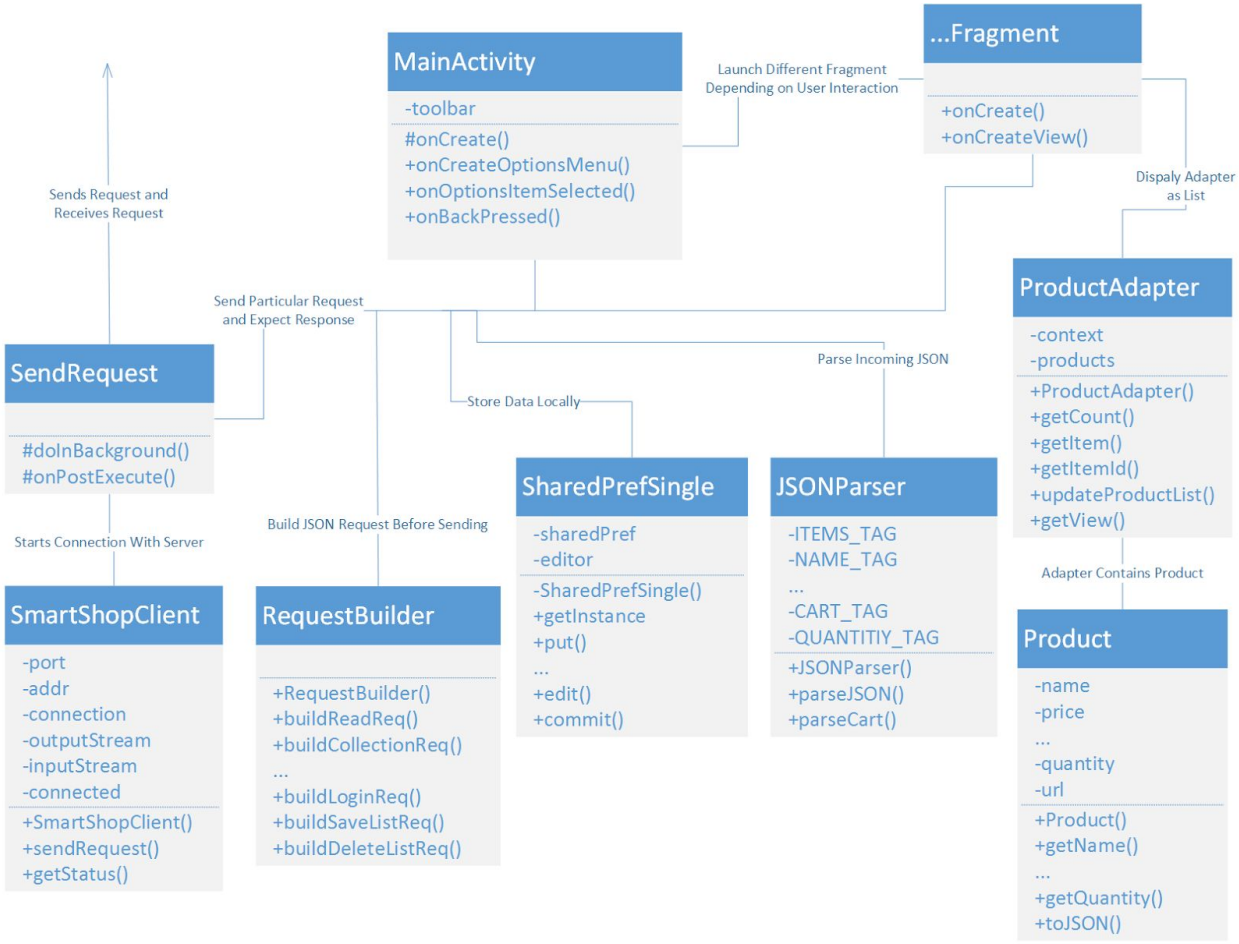
Main Server Class Diagram



Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

Android

Android Class Diagram



Android Overview

When the Android application starts, the MainActivity will be launched and serve as the starting point of all other operations of the application. Depending on what the user selects in the UI, different fragments (...Fragment) will be launched. For example, if the user presses on the login button, the fragment corresponding to login will be launched (LoginFragment). There are a variety of different classes to support the handling of data within the application; these include RequestBuilder, SharedPrefSingle, and JSONParser. RequestBuilder handles building JSON requests when the application has been prompted by the user to perform actions that need to query the server. SharedPrefSingle is a singleton class that handles the storage of data in local storage. JSONParser handles parsing any incoming or stored JSON strings into useful data.

When the application is at a stage where a request needs to be sent to the server, the SendRequest AsyncTask will be called. To establish the connection, SendRequest will call the SmartShopClient to connect to the correct ip and port before the request is sent. When the data arrives, the JSON will be parsed into Product objects. And in order to finally display the search

resulting data to the user, a ProductAdapter containing the list of Products will be created, and subsequently displayed by the fragment that made the request.

Web

Overview

The web backend consists of a reverse proxy server and a web server. The reverse proxy server listens on port 443, the standard port for secure HTTPS connections. It also listens on port 80, redirecting all HTTP requests to HTTPS. Upon receiving a client request from the internet (e.g. a user attempts to visit the website on their browser), the reverse proxy server redirects the request to the web server, which is listening on port 3000.

Once the web server receives an HTTPS request, it first determines the request's method (or verb) and the resource that the method wishes to act upon. For example, a user visiting https://www.checkedout.ca/ would result in a GET request (the method) for the home page of the website (the resource). The task of deciphering the type of request received is done by the Main module. The Main module then calls the appropriate handler function in the Handler module to take care of the request.

For some requests, the Handler module does not need any data from the main server (e.g when the user visits the home page). For such requests, the Handler module can generate the appropriate HTTPS response directly and send this back to the client (through the reverse proxy). For requests that need data from the main server (e.g. when the user searches for an item), the Handler module calls a factory function in the Messenger module to create and send a request to the main server for data.

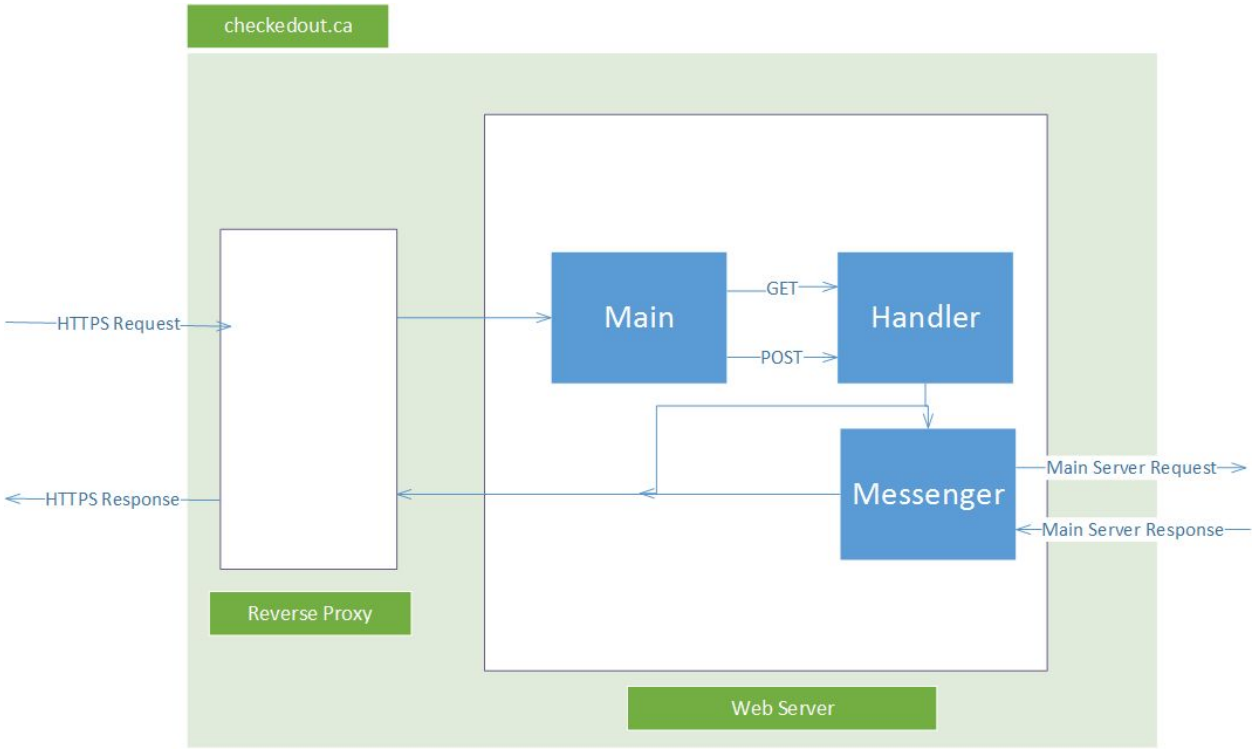
When the factory function in the Messenger module is called, it creates the appropriate request object which depends on the type of HTTPS request initially received by the web server (this information is provided by the Handler module). The Messenger module then opens a TCP socket connection to the main server and sends the request object. The main server receives the request object and eventually sends back a response. During this process, the Messenger module listens on the socket for incoming data. Once the response is fully received from the main server, the Messenger module closes the socket, deciphers the response, and outputs an appropriate HTTPS response back to the client (again, through the reverse proxy).

The web server is responsible for serving the website for our application. Thus, a typical HTTPS response from the web server includes an HTML string that represents the web page that should be displayed to the client. The HTML string is generated by rendering Pug files, which behave like HTML “templates” that allow for web page elements to be defined dynamically (e.g. the search results for an item). Additional details about Pug are explained in the GUI section.

As mentioned previously, the web server utilizes Node.js and various Javascript frameworks for its functionalities. Because Node.js is asynchronous and event driven, multiple users can connect to the web server concurrently without fear of deadlocks or sharing of data.

Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

Web Diagram

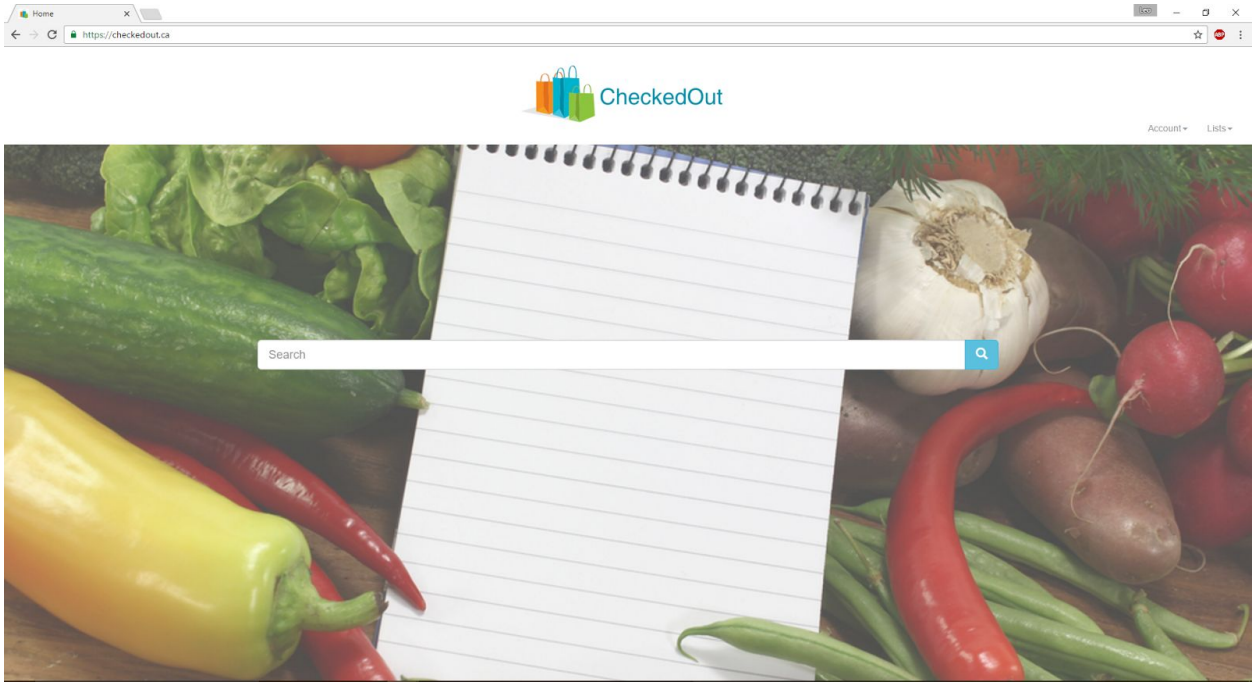


GUI

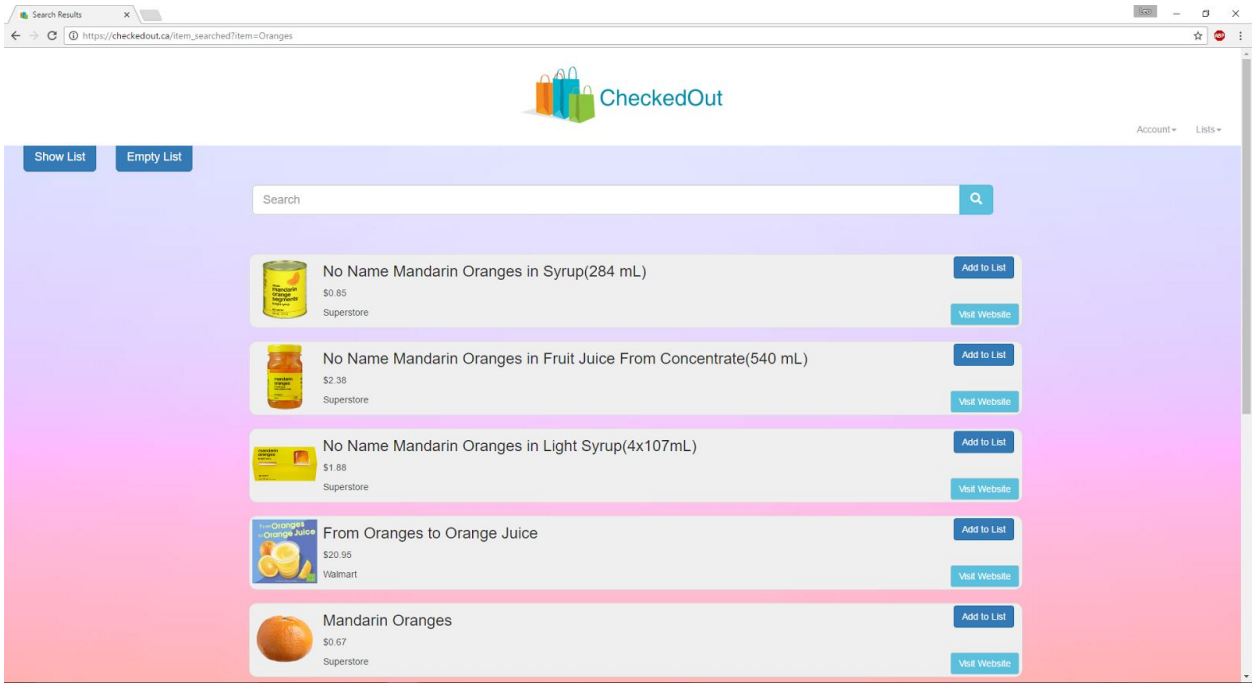
The application includes two main user interfaces: the website user interface and the mobile user interface. Included below are images of the user interfaces, with the mobile user interface being a simple, condensed version of the website user interface. The mobile interface is based on Google's material design principles, and is defined through XML and Java. The website user interface is designed using Pug (HTML), CSS, and Javascript. Bootstrap was also utilized for the website UI as it provided a solid base of CSS/Javascript to build upon. It was used in the development of features such as the navigation bar, and the search bar. Pug, previously known as Jade, is an HTML templating engine which allows our webpages to be both dynamic, and reusable. For example, our search bar is in its own Pug file, and is simply "included" on any desired pages. This provided us with a way of increasing code reusability, as it saved us from copying and pasting code for multiple pages.

Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

Website - Dashboard View

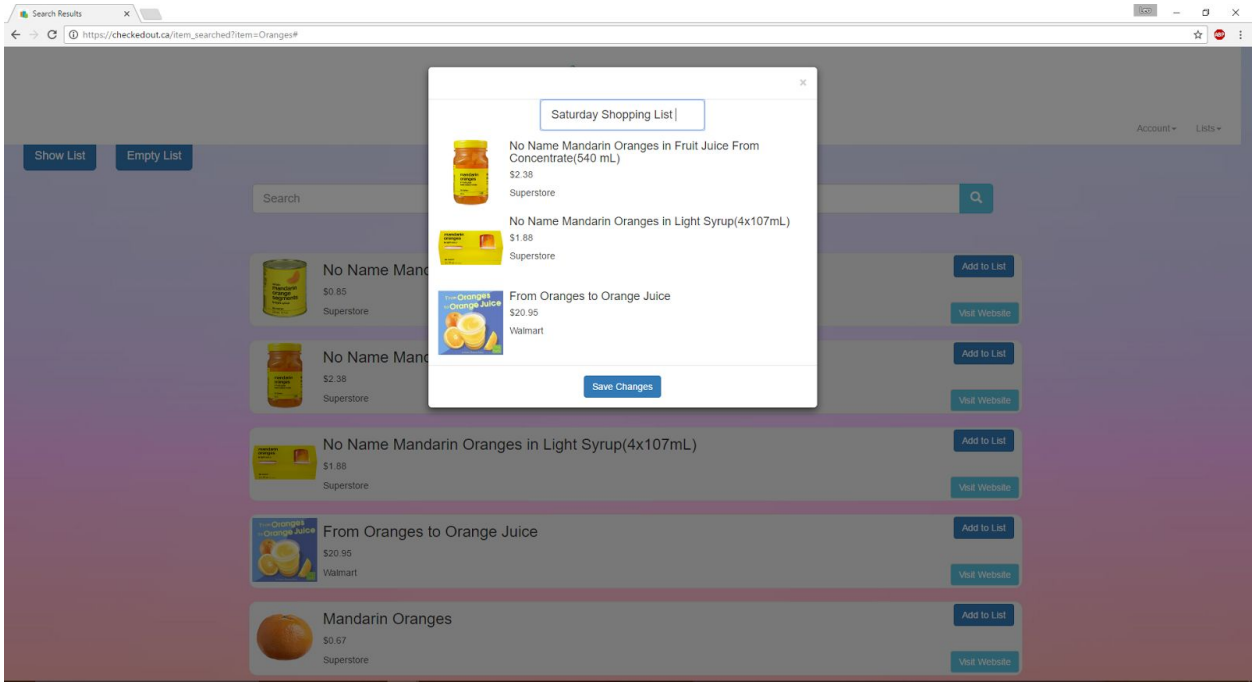


Website - Search Results

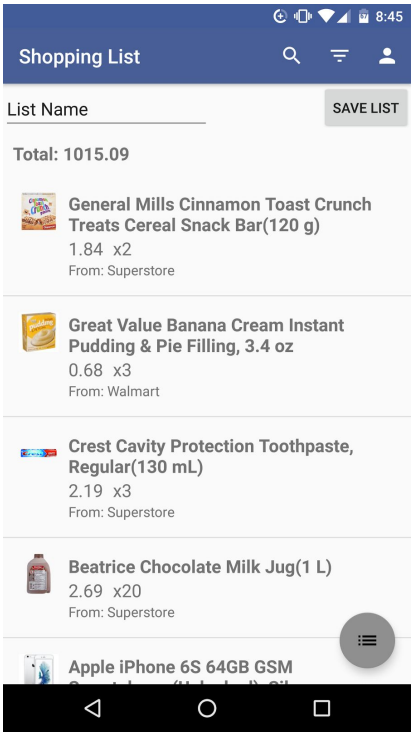


Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

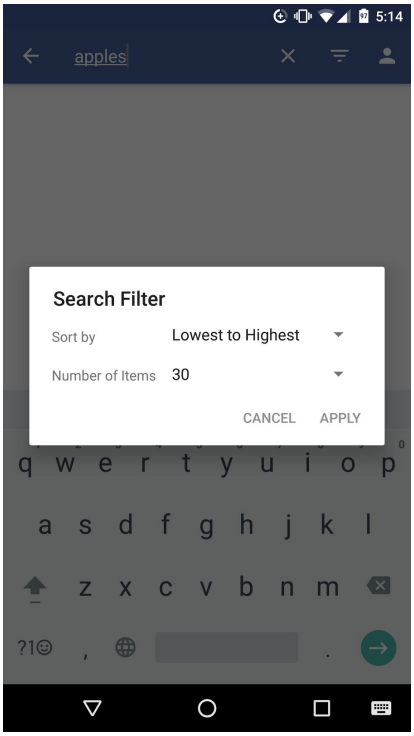
Website - Shopping List View



Android – Shopping List View

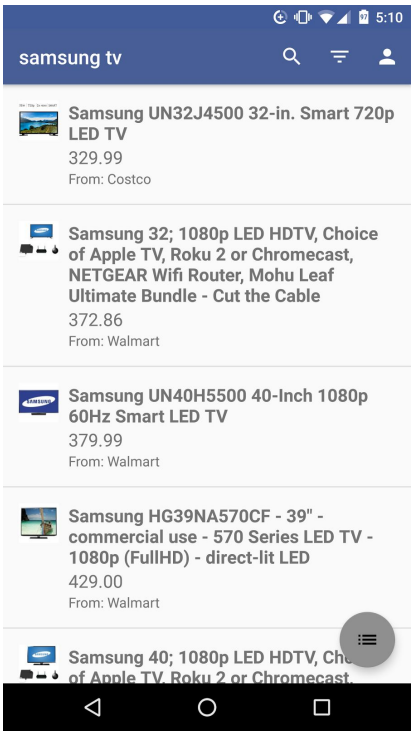


Android – Search Options



Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

Android – Search Results



The app screenshots shown above indicate some of the features in our product. The “Shopping Lists” page is intended to be a separate page where a user can view their current shopping list. Once logged in, there will be additional options to view or edit previous lists, as well as create new lists and saving them online. The “Search Filter” allows users to specify the manner of which the search results should be sorted, for example the price from “lowest to highest” or “highest to lowest”. The user is also given the option to choose the amount of results that should be returned once the search is completed.

Validation

Validation has been an integral part of our planning progress, as we sought continuous feedback from our family and friends. Our group met after every session and discussed the results of the feedback, including how we can properly change our design to fit the requirements of different users. Thus, our design has evolved from our initial draft to a streamlined version that has many elements that are inspired from user feedback.

One example is that some users have said that while our Android application is well designed – it is very plain and some extra information is preferred. Specifically, the home page of the application should not simply be empty, and instead should display some relevant information such as recommended and popular items. In addition, many users are confused about the lack of interactivity with the actual items, as tapping them does not bring up any more detailed information about the items.

When the same users tried our web application, they said the same thing as they did with the Android application. However, they also said that the design of the website needed some improvements, such as color contrast and things like making the UI easier to navigate.

Overall, we feel that validation was a critical part of our project, as it allowed us to make changes to our design based on the feedback from users.

Verification

We plan to verify our servers and database using stress testing that simulates scenarios such as weekends where there may be many end-users that wish to find the cheapest price for certain items. This will be achieved by integrating Locust.io to simulate hundreds or thousands of user requests, replicating a timeframe in which many users would want to access our application. The stress testing process will allow us to see how our system reacts under high stress, and these reactions could range from handling it perfectly to crashing entirely.

Our testing gave rise to the multi-threading system that we implemented. The need for this multi-threaded system is so that we are able to asynchronously handle requests concurrently from many different users. This way, many different users can access our service and be given reasonable response times.

Some other tests we have implemented, such as unit and system tests, are described in full detail in our Testing Document. Our plan is to run the both the stress, unit, and system tests after each important change to our project, as we wish to verify the correctness of our system to ensure that all functionality works as intended.

Developer Section

Source code is available at: https://github.com/BenjaminLang/cpen_321

Installation Guide

Because the installation for Windows and Ubuntu are very similar, below is an installation guide for both versions. However, please make sure to download and install the correct version of each dependency.

1. Install the following dependencies:
    a. Python3.5.2
    b. MongoDB
    c. PyMongo
    d. bson
    e. BeautifulSoup4
    f. Locust.io
    g. Node.js
    h. Npm
    i. JDK (>= 7)

   j. Android Studio
   k. Picasso
2. Pull the latest version of the master branch
3. Run MongoDB on the terminal with “mongod --config “PATH_TO_CONFIG””
   a. Config is located at Server/docs/mongod_ver.cfg
   b. Use mongod_win.cfg if on a Windows machine
   c. Use mongod_u.cfg if on an Ubuntu based machine
4. To run the different tests:
   a. Main Server
      i. Run the main server with “python3 Server/src/main.py”
      ii. Run the various tests with:
         1. “python3 Server/test/list_test.py”
         2. “python3 Server/test/read_write_test.py”
         3. “python3 Server/test/user_test.py”
      iii. To run the “stress_test.py”, refer to Locust.io documentation
         1. http://docs.locust.io/en/latest/running-locust-distributed.html
   b. Webserver
      i. In Web/, run “npm install” to install Node.js dependencies
      ii. Run “npm run postinstall” to configure the testing environment
      iii. Run all tests with: “npm run tests”
   c. Android
      i. Run all JUnit tests

Tests

Main Server:

- stress_test.py - stress testing source file for Locust Framework
- list_test.py - list functionality test suite for main server
- read_write_test.py - read/write functionality test suite for main server
- user_test.py - user operations functionality test for main server

Webserver:

- all tests in Web/tests

Android:

- all tests in android/tests folder

Design Patterns

- Singleton
- Factory

Benjamin Lang; Amrit Kooner; Leo Belanger
Ryan Liu; Andy Wong; John Sun

- Observer

Source Code Directory Structure

Android
    SmartShopper
        app
            src
Server
    src
    test
Web
    public
        css
        html
        js
    tests
    views