Smart Shopping List

The Chaincoders

Combined SRS-SDD-Test Document

Zeynep Akkalyoncu - 2035541
Arda Efe Okay - 1937333
Erinç Argımak - 1941772

Table of Contents

1. System Requirements

a. Specific Requirements

i. Functionalities and Operations

The "Smart Shopping List" application is a basic shopping list application enhanced with machine learning, artificial intelligence and national language processing techniques. Using "Smart Shopping List", you can create your online shopping lists and share them in app or via Facebook. The app can recommend you some products according to your habits and preferences. In addition to these, the Virtual Assistant - ChatBot can help you in app to find some specific products or just operate the app.

You can find the cheapest real items from real markets and add them to lists. Also, at market you can mark them as bought.

ii. User Interfaces

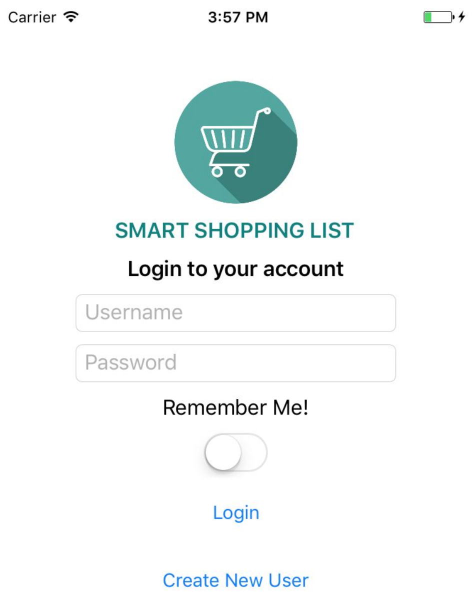


Figure 1: Login page

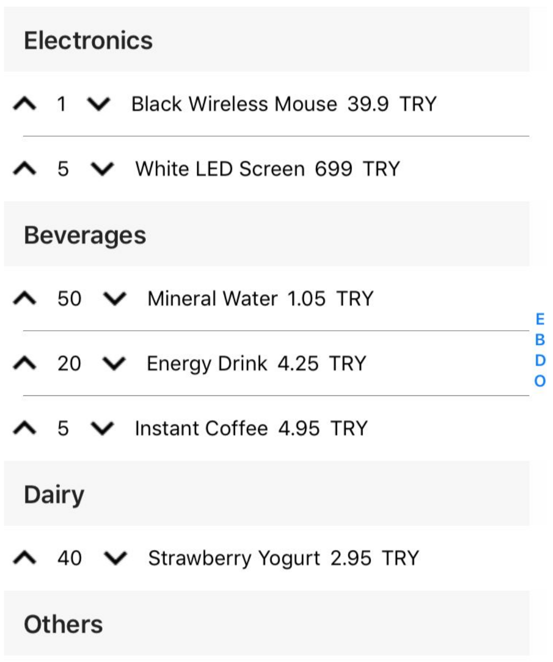

Electronics
1 Black Wireless Mouse 39.9 TRY
5 White LED Screen 699 TRY
Beverages
50 Mineral Water 1.05 TRY
20 Energy Drink 4.25 TRY
5 Instant Coffee 4.95 TRY
Dairy
40 Strawberry Yogurt 2.95 TRY
Others
E
B
D
O

Figure 2: Example shopping list view

The user shall see the login page in Figure 1 when they first open the application. Other main pages include the list page, shown in Figure 2, user page, virtual assistant page, new item page, etc. The user interfaces shall be simple and intuitive enough for anyone to operate; all the pages are designed with this goal in mind. All of the pages can be accessed with a single click from the master view in the master-detail navigation model. Longer scroll/list views have a quick access control on the right with the initial of a subcategory name pointing to the respective subcategory.

iii. Software Interfaces

The “Smart Shopping List” application can run on Android and iOS operating systems. So, user’s mobile device should have one of these operating systems.

iv. Hardware Interfaces

Since the application must remain connected to the Internet for real-time synchronization, the user’s mobile device must have a functioning wireless unit, and an accessible connection device nearby (e.g: modem).

v. Communication Interfaces

“Smart Shopping List” mobile application uses the HTTP protocol for communication over the Internet with the Django backend hosted on the Azure virtual machine, and TCP/IP protocol suite for intranet communication.

vi. Memory Constraints

The user’s mobile device must have at least 54 MB spare storage to install and have the application run smoothly. This space is required to accommodate the application executable as well as embedded images.

vii. Limitations

Due to time limitations of project and workload, our product dataset include only carrefoursa market. This may populated with other scrapper and crawler tools.

“Smart Shopping List” application can run on iOS 10 or Android Api Level 15 (4.0.3) and further.

Due to Chatbot - NLP part, app language is limited with English.

b. Project Plan

We had initially planned to complete the basic shopping list application with full capabilities (i.e: creation of lists tied to users in the backend) in the first four sprints. (1 sprint = 3 weeks) We had originally intended to develop our application primarily in Android, and then later port it to iOS as a bonus feature. After we attended a seminar at the department by a Xamarin representative, we decided to switch over to Xamarin Forms so that we could pursue both more quickly.

At least two weeks of website crawling and data scraping had to be performed in advance in order to gather sufficient data to populate the database significantly, which would be used for list item operations on the client-side, and later as machine learning training data. This estimated amount of time was enough despite certain anticipated challenges (e.g: running the script at a controlled rate to avoid getting banned) although we had to repeat this process a few more times throughout the year due to other technical issues.

In the meantime, we started constructing the backend using Azure's Mobile App Services. While this service was practical in communicating from the Xamarin application with the database, we found it was not flexible enough to build our own web services - recommendation systems and virtual assistant - that lied at the core of our project. We struggled for a while during the search for a better option, at the end of which we successfully started constructing a Django RESTful API, which greatly accelerated our progress from that point on.

In the beginning, we estimated sixteen man-months' worth of work for the development of recommendation systems, and nine for virtual shopping assistant. In other words, we allocated the majority of our remaining time following the initial mobile application development period to these web services. Because we spent more time on other tasks than planned earlier, we had to complete these tasks at a faster rate. Finally, we managed to finish all the relevant tasks in the very last sprint in spite of the delay.

Other remaining tasks such as social media integration and integration, particularly those regarding custom iOS and Android controls, were tricker than we thought, and couldn't be finished in the expected 2-3 three weeks.

c. Use Case Diagrams

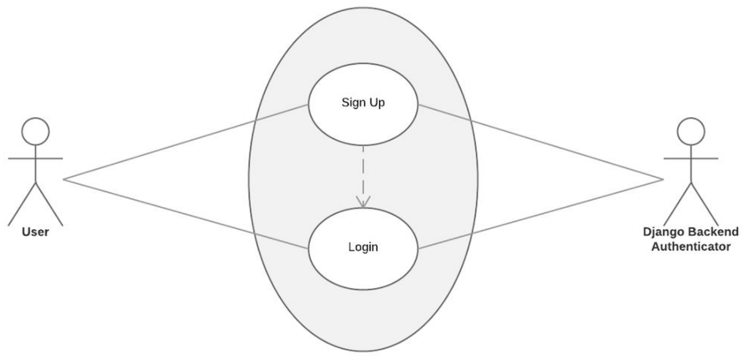


Figure 3: Use Case Diagram 1: Login Operations

If a user don’t have a ChainCoders Account, s/he can it with ChainCoders App. Also, User can login with ChainCoders Account or Facebook account to our app.

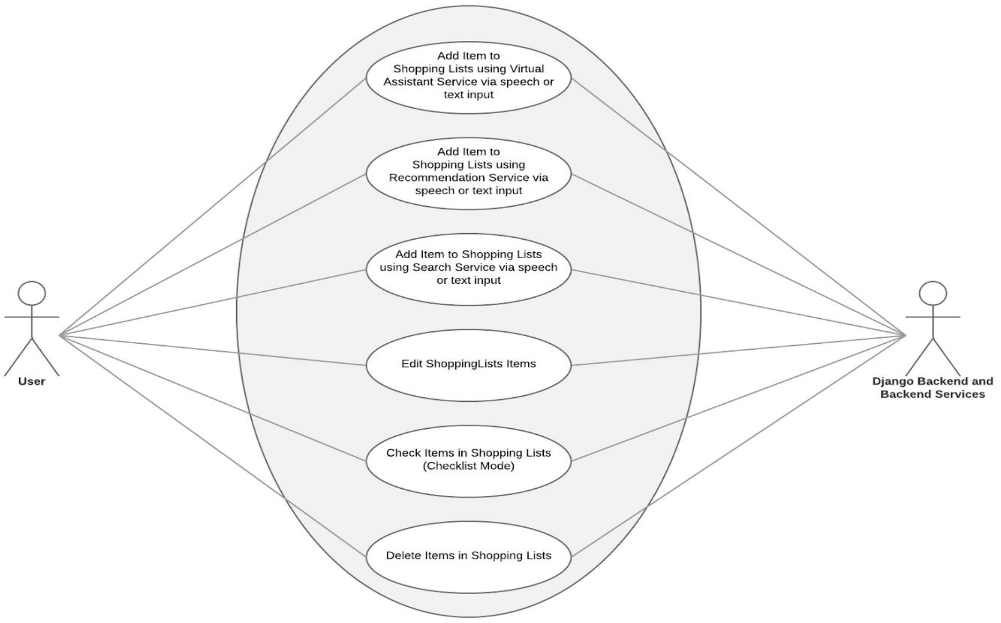

Add Item to Shopping Lists using Virtual Assistant Service via speech or text input
Add Item to Shopping Lists using Recommendation Service via speech or text input
Add Item to Shopping Lists using Search Service via speech or text input
Edit ShoppingLists Items
Check Items in Shopping Lists (Checklist Mode)
Delete Items in Shopping Lists
User
Django Backend and Backend Services

Figure 4:. Use Case Diagram 2: Shopping List Operations

Using our app, users can create new shopping lists, edit existing ones and browse them. There are tags that lists can have. Users can use them to identify their shopping lists. User can share their shopping lists with other to collaborate with them

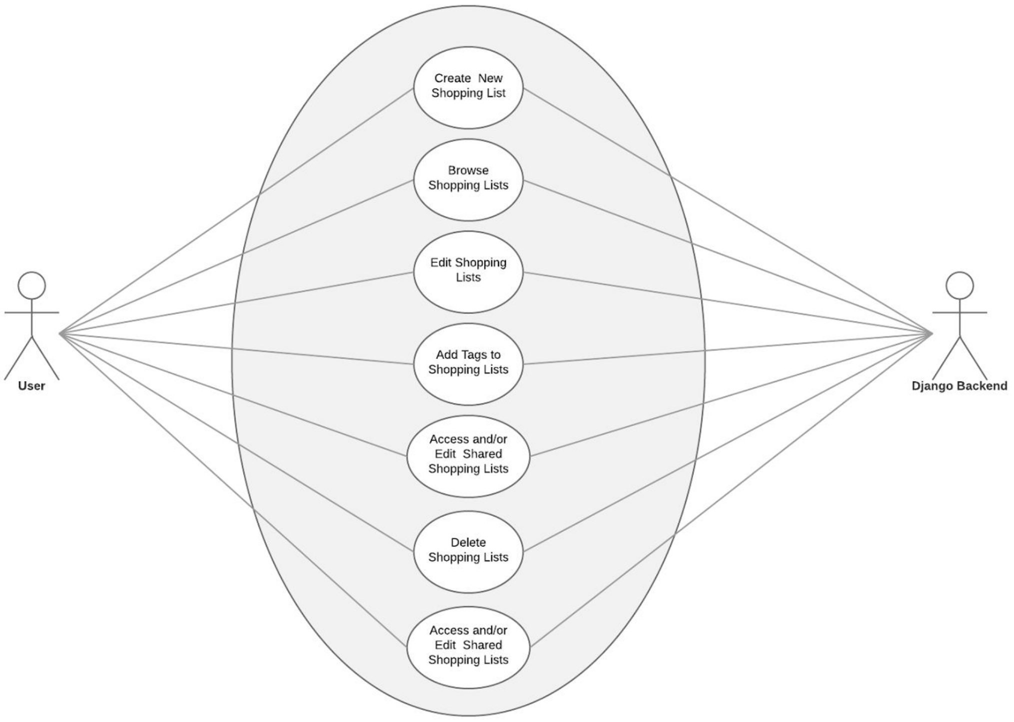

Create New Shopping List
Browse Shopping Lists
Edit Shopping Lists
Add Tags to Shopping Lists
Access and/or Edit Shared Shopping Lists
Delete Shopping Lists
Access and/or Edit Shared Shopping Lists
User
Django Backend

Figure 5: Use Case Diagram 3: Item Operations

There are 3 ways to add a product to shopping lists. These are using virtual assistant, using recommendation system and using search service. At both way, user can interact with service with speech or text and can get response with speech and text. User can change the quantity of shopping list products and you can delete them. In addition to these, user can mark the items like hand written shopping lists.

2. System Design

a. Module Structure

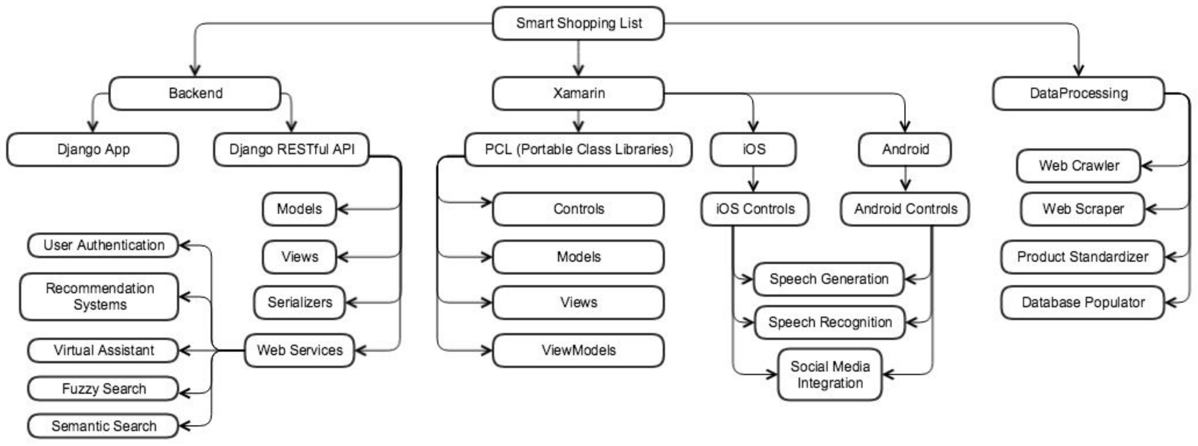


Figure 6: Module structure of mobile application "Smart Shopping List"

b. Git Source Code Structure

The structure under our master branch is shown below. Auto-generated configuration and training/test data files are not included. Note that actual file names are replaced by their explanations in italic for brevity.

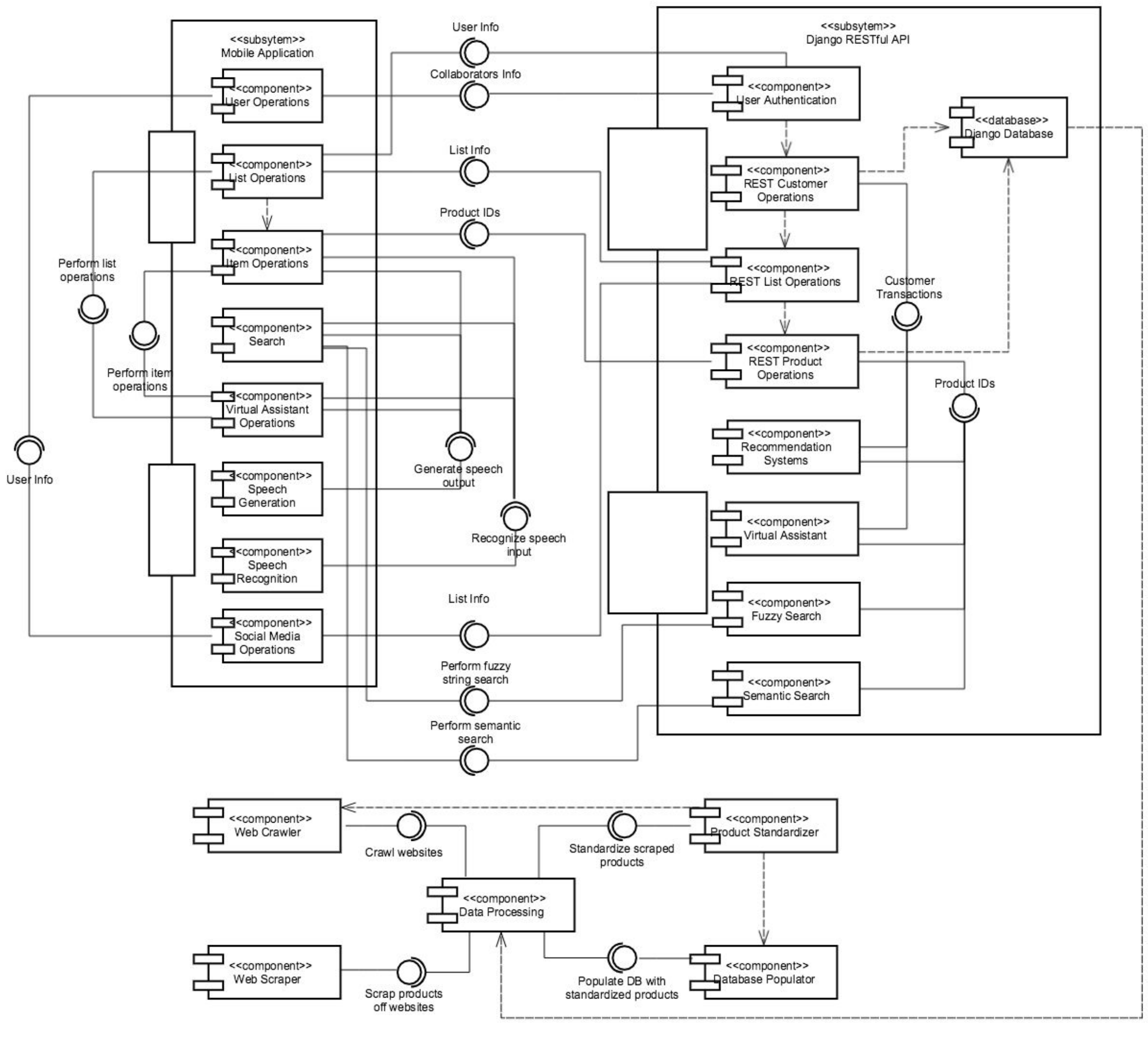
SmartShoppingList

- Backend
    - shoppinglist
        - *Django app configuration*
    - api
        - *Django RESTful API implementation*
        - Chatbot
            - Conversational
                - *deep learning chatbot model*
                - *chatbot logic implementation*
        - Recsys
            - content_based
                - *content based recommendation implementation*
            - collaborative
                - *item and user based recommendation implementation*
            - *hybrid recommendation implementation*
- Data
    - ScraperCrawler
        - *website scraper/crawler C++ source code*
    - *product standardizer and database populator Python scripts*
- Xamarin
    - chaincoders
        - *application configuration and main file*
        - Controls
            - *custom visual, speech-text-speech, etc. control implementations*
        - Models
            - *JSON classes for backend connection and other client-side models*
        - ViewModels
            - *C# source code to define display rules for data*
        - Views

            - *C# source code and XAML declaration files for mobile application views*
    - iOS
        - *iOS application configuration and main file*
        - Controls
            - *custom iOS implementations for relevant controls defined in shared folder*
    - Droid
        - *Android application configuration and main file*
        - Controls
            - *custom Android implementations for relevant controls defined in shared folder*

---

c. Component Diagram



<<subsytem>> Mobile Application
<<component>> User Operations
<<component>> List Operations
<<component>> Item Operations
<<component>> Search
<<component>> Virtual Assistant Operations
<<component>> Speech Generation
<<component>> Speech Recognition
<<component>> Social Media Operations
Perform list operations
Perform item operations
User Info
Collaborators Info
List Info
Product IDs
Generate speech output
Recognize speech input
List Info
Perform fuzzy string search
Perform semantic search
<<subsytem>> Django RESTful API
<<component>> User Authentication
<<database>> Django Database
<<component>> REST Customer Operations
<<component>> REST List Operations
<<component>> REST Product Operations
Customer Transactions
Product IDs
<<component>> Recommendation Systems
<<component>> Virtual Assistant
<<component>> Fuzzy Search
<<component>> Semantic Search
<<component>> Web Crawler
Crawl websites
Standardize scraped products
<<component>> Product Standardizer
<<component>> Data Processing
<<component>> Web Scraper
Scrap products off websites
Populate DB with standardized products
<<component>> Database Populator

Figure 7: Component diagram of mobile application “Smart Shopping List”

d. Deployment Diagram

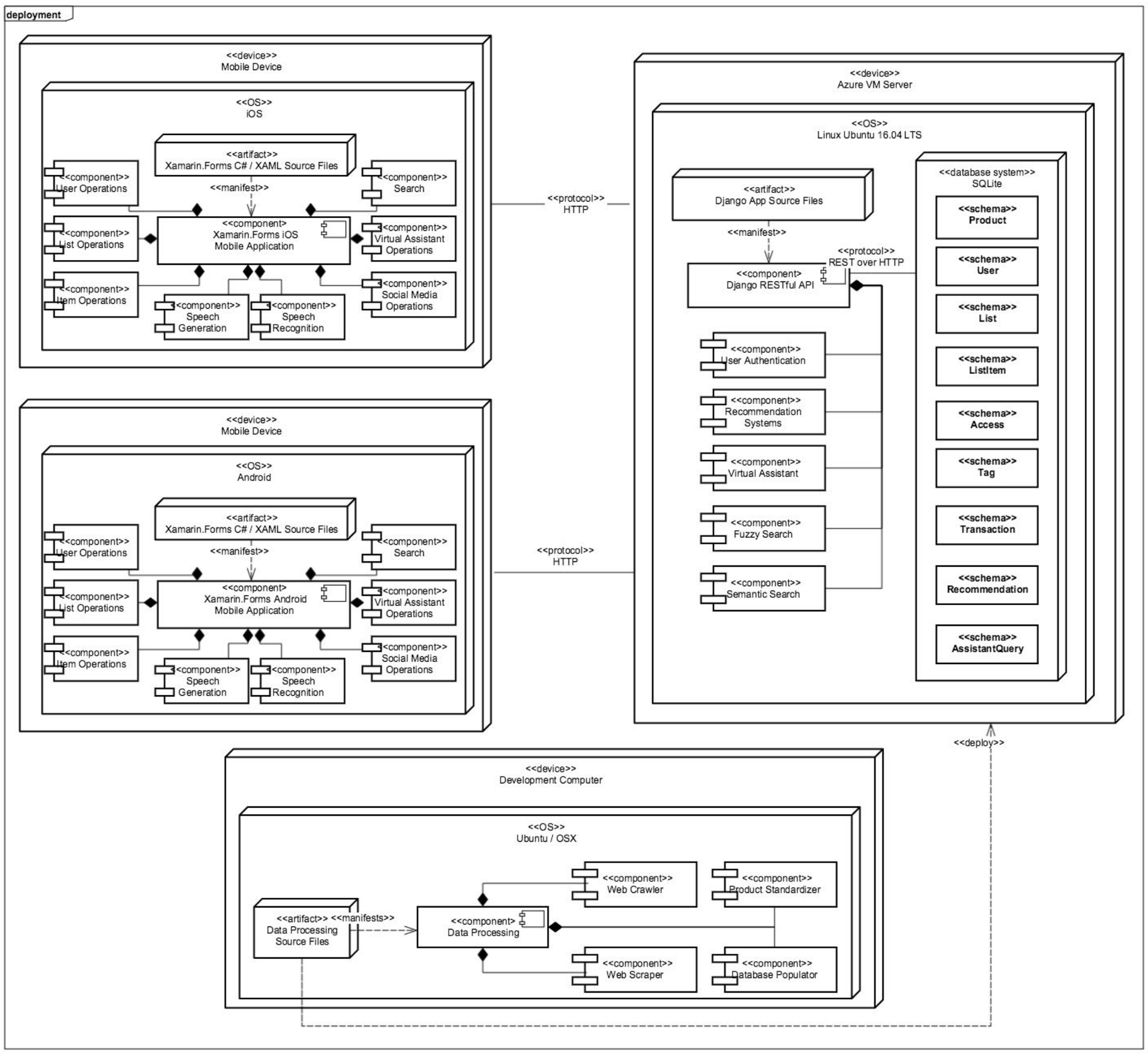

deployment
<<device>> Mobile Device
<<OS>> iOS
<<artifact>> Xamarin.Forms C# / XAML Source Files
<<manifest>>
<<component>> Xamarin.Forms iOS Mobile Application
<<component>> User Operations
<<component>> List Operations
<<component>> Item Operations
<<component>> Search
<<component>> Virtual Assistant Operations
<<component>> Social Media Operations
<<component>> Speech Generation
<<component>> Speech Recognition
<<device>> Mobile Device
<<OS>> Android
<<artifact>> Xamarin.Forms C# / XAML Source Files
<<manifest>>
<<component>> Xamarin.Forms Android Mobile Application
<<protocol>> HTTP
<<protocol>> HTTP
<<device>> Azure VM Server
<<OS>> Linux Ubuntu 16.04 LTS
<<artifact>> Django App Source Files
<<manifest>>
<<component>> Django RESTful API
<<protocol>> REST over HTTP
<<component>> User Authentication
<<component>> Recommendation Systems
<<component>> Virtual Assistant
<<component>> Fuzzy Search
<<component>> Semantic Search
<<database system>> SQLite
<<schema>> Product
<<schema>> User
<<schema>> List
<<schema>> ListItem
<<schema>> Access
<<schema>> Tag
<<schema>> Transaction
<<schema>> Recommendation
<<schema>> AssistantQuery
<<deploy>>
<<device>> Development Computer
<<OS>> Ubuntu / OSX
<<artifact>> Data Processing Source Files
<<manifests>>
<<component>> Data Processing
<<component>> Web Crawler
<<component>> Web Scraper
<<component>> Product Standardizer
<<component>> Database Populator

Figure 8: Deployment diagram of mobile application “Smart Shopping List”

3. Testing

a. Test Plan and Scenarios

Testing of the client side application was planned to be UI Testing mostly, checking if all views function as intended, i.e. leading to correct pages or information. Test plan included Android application testing using "Xamarin.UITest" package, which runs tests on dedicated devices via "NUnit" package test adapter. An independent test sub-project was created. Test Plan included checking for:

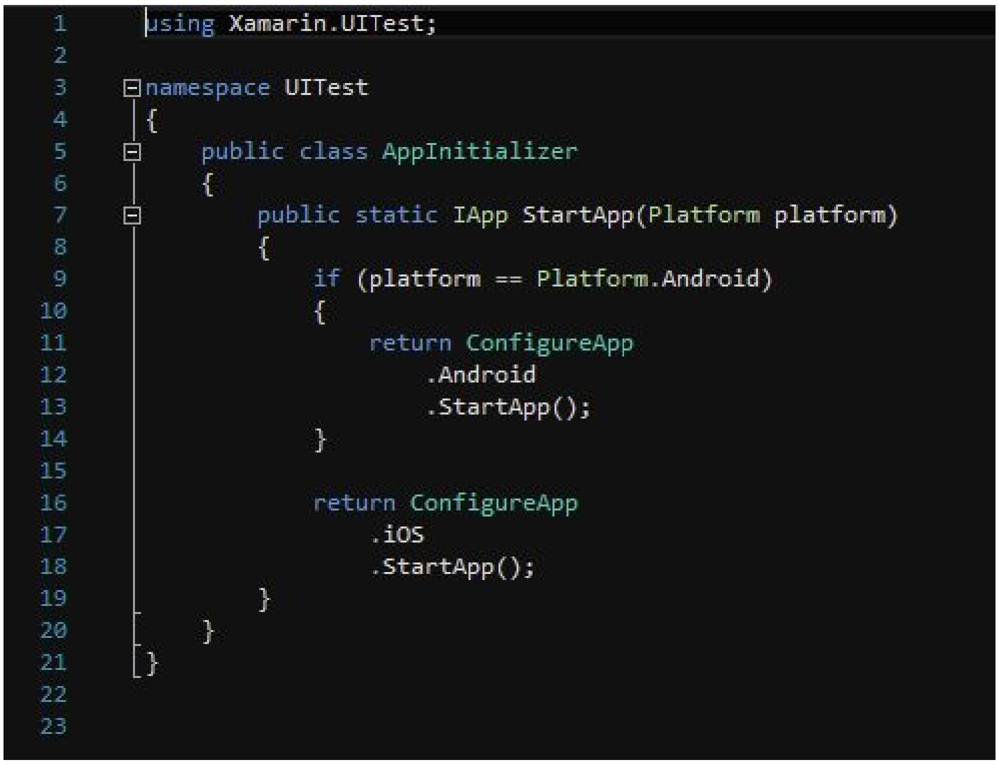
- Application Login Page Mechanism
- Application List Page
- New Item screen
- Side Menu Functionality

Test scenarios included:

- User opens the app.
- User enters credentials and presses the login button.
- User picks a list from menu.
- User picks an item from the list.
- User invokes side menu view.

b. Git Testing Code Structure

Under the master branch, the test project is located in the "Xamarin" folder and is included in the Application solution as a project. It is built and deployed with the project. The references of projects which correspond to the tested platforms are linked into the test project. The test code consists of two parts:

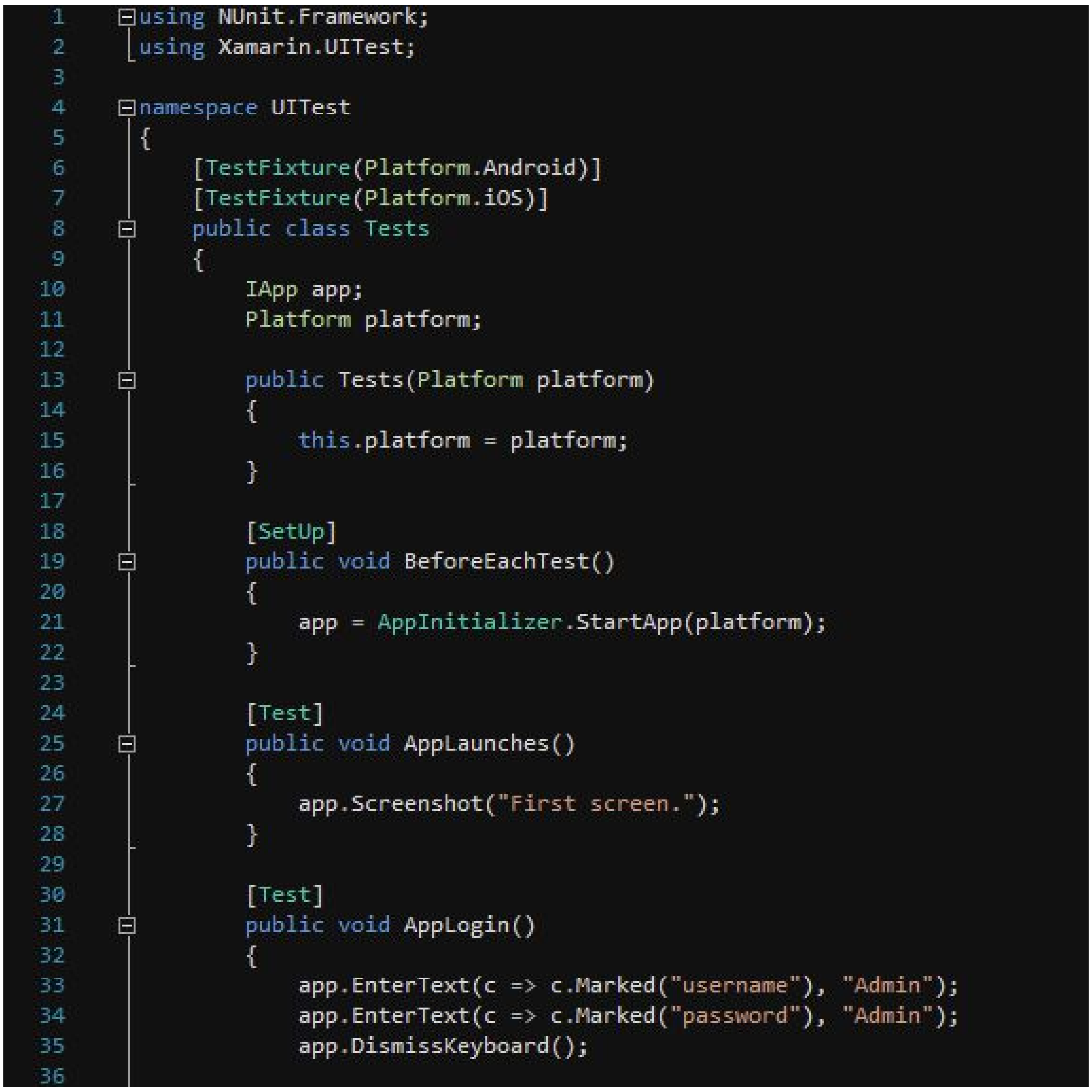


```
using Xamarin.UITest;

namespace UITest
{
    public class AppInitializer
    {
        public static IApp StartApp(Platform platform)
        {
            if (platform == Platform.Android)
            {
                return ConfigureApp
                    .Android
                    .StartApp();
            }

            return ConfigureApp
                .iOS
                .StartApp();
        }
    }
}
```

Figure 9: App Initializer code



```
using NUnit.Framework;
using Xamarin.UITest;

namespace UITest
{
    [TestFixture(Platform.Android)]
    [TestFixture(Platform.iOS)]
    public class Tests
    {
        IApp app;
        Platform platform;

        public Tests(Platform platform)
        {
            this.platform = platform;
        }

        [SetUp]
        public void BeforeEachTest()
        {
            app = AppInitializer.StartApp(platform);
        }

        [Test]
        public void AppLaunches()
        {
            app.Screenshot("First screen.");
        }

        [Test]
        public void AppLogin()
        {
            app.EnterText(c => c.Marked("username"), "Admin");
            app.EnterText(c => c.Marked("password"), "Admin");
            app.DismissKeyboard();

```

Figure 10: Test Cases (Partial sample)

As a tree mode:
SmartShoppingList (Referring to the one in Git Source Code Structure part)

- Xamarin
    - chaincoders
        - ...
    - iOS
        - ...
    - Droid
        - …
    - UITest
        - AppInitializer
        - Tests
        - References

c. Test Results

Test results consist of several screenshots that correspond to the screens at that point of testing. After production of these images, the tester is responsible for checking them against the intended screens and report. Upon testing, we have confirmed that the views function as intended.

4. References

This document was written in accordance with the IEEE Standards as dictated in the following report: “ISO/IEC/IEEE 29148:2011 Systems and software engineering - Life cycle processes - Requirements Engineering”.